KAYPRO

Microsoft™ BASIC

microsoft BASIC user's guide

for 8080/8085 and Z80
Microprocessors and
CP/M Operating System

LIMITED WARRANTY

MICROSOFT, INC., shall have no liability or responsibility to purchaser or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT, INC. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.

8108A-530-00
30F14B

Introduction

BASIC Release 5 from Microsoft is the most extensive implementation of BASIC available for microprocessors. It meets the requirements for the ANSI subset standard for BASIC, and supports many unique features rarely found in other BASICs. In addition, Microsoft BASIC has sophisticated string handling and structured programming features that are especially suited for applications development. Microsoft BASIC gives users what they want from a BASIC -- ease of use plus the features that make a micro perform like a minicomputer or large mainframe.

In 1975, Microsoft wrote the first BASIC interpreter for the microcomputer. Today Microsoft BASIC, with over 750,000 installations in over 20 operating environments, is recognized as the industry standard. It's the BASIC you'll find on all the largest-selling microcomputers. Many users, manufacturers, and software houses have written application programs in Microsoft BASIC.

Microsoft

Welcome to the Microsoft family of products.

Microsoft, Inc. is recognized as the leader in microcomputer software. Microsoft BASIC interpreter, in its several versions, has become the standard high-level programming language used in microcomputers. Microsoft, Inc. continues to supply consistently high-quality software which sets the standard for software quality for all types of users.

For more information about other Microsoft products, contact:

Microsoft, Inc.
10700 Northup Way
Bellevue, WA 98004

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Contents

1. Four variable types: Integer (±32767), String (up to 255 characters), Single Precision Floating Point (7 digits), Double Precision Floating Point (16 digits)

2. Trace facilities (TRON/TROFF) for easier debugging

3. Error trapping using the ON ERROR GOTO statement

4. PEEK and POKE statements to read and write any memory location

5. Automatic line number generation and renumbering, including referenced line numbers

6. Arrays with up to 8 dimensions

7. Boolean operators OR, AND, NOT, XOR, EQV, IMP

8. Formatted output using the complete PRINT USING facility, including asterisk fill, floating dollar sign, scientific notation, trailing sign, comma insertion

9. Direct access to I/O ports with the INP and OUT functions

10. Extensive program editing facilities via EDIT command and EDIT mode subcommands.

11. Assembly language subroutine calls (up to 10 per program) are supported.

12. IF/THEN/ELSE and nested IF/THEN/ELSE constructs

13. Disk BASIC supports variable length random and sequential disk files with a complete set of file manipulation statements: OPEN, CLOSE, GET, PUT, KILL, NAME, MERGE

NOTE

Features vary from one implementation of Microsoft BASIC to the next. See Chapter 1 of this manual for an exact description of the features in your implementation of Microsoft BASIC.

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Using these Manuals

The information in the documents you received in this package is divided into reference information and user information.

The Microsoft BASIC Reference Manual contains descriptions of all the features of Microsoft BASIC. The Reference Manual contains no information that is either implementation specific (that is, applies to a particular microprocessor), or operating system specific.

The Microsoft BASIC User's Guide contains all of the implementation-specific and operating-system specific information. This information includes telling you which features of Microsoft BASIC are included and which are excluded from your implementation, telling you how your implementation changes the format and use of some features, and telling you how your operating system affects some features and operations of Microsoft BASIC.

Use the Reference Manual for details of the features. Use the User's Guide to see if a feature differs from its description in the Reference Manual. Also, use the User's Guide for hints about disk I/O and calling assembly language subroutines into your BASIC programs.

The Microsoft BASIC Reference Book is a quick reference guide to the features and their syntax. The Reference Book includes all the features described in the Reference Manual, but does not contain information that is implementation-specific or operating system-specific.

[illegible]

Syntax Notation

The following notation is used throughout this manual in descriptions of command and statement syntax:

[ ] Square brackets indicate that the enclosed entry is optional.

< > Angle brackets indicate user entered data. When the angle brackets enclose lower case text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose upper case text, the user must press the key named by the text; for example, <RETURN>.

{ } Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.

... Ellipses indicate that an entry may be repeated as many times as needed or desired.

CAPS Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

CHAPTER 1

FEATURES INCLUDED IN THIS IMPLEMENTATION

All statements, commands, and functions described in the Microsoft BASIC Reference Manual are implemented unless listed below. Some of the statements, commands, or functions may be affected by this particular implementation. Read Chapter 2, Langauge Differences for This Implementation.

The following features are not included in the Standard Disk Version of Microsoft BASIC:

CLOAD CSAVE

CHAPTER 2

LANGUAGE DIFFERENCES FOR THIS IMPLEMENTATION

The features in your version of Microsoft BASIC are as described in the Microsoft BASIC Reference Manual.

CHAPTER 3

CONVERTING PROGRAMS TO MICROSOFT BASIC

If you have programs written in a BASIC other than Microsoft BASIC, some minor adjustments may be necessary before running them with Microsoft BASIC. Here are some specific things to look for when converting BASIC programs.

3.1 STRING DIMENSIONS

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the Microsoft BASIC statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for Microsoft BASIC string concatenation.

In Microsoft BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	Microsoft BASIC
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASIC	Microsoft BASIC
A$(I)=X$	MID$(A$,1,1)=X$
A$(I,J9=X$	MID$(A$,I,J-I+1)=X$

3.2 MULTIPLE ASSIGNMENTS

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. Microsoft BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

3.3 MULTIPLE STATEMENTS

Some BASICs use a backslash (\) to separate multiple statements on a line. With Microsoft BASIC, be sure all statements on a line are separated by a colon (:).

3.4 MAT FUNCTIONS

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

CHAPTER 4

MICROSOFT BASIC DISK I/O

Disk I/O procedures for the beginning BASIC user are examined in this appendix. If you are new to BASIC or if you're getting disk related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements for filenames. The CP/M operating system will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE or LOAD command.

4.1 PROGRAM FILE COMMANDS

Here is a review of the commands and statements used in program file manipulation.

SAVE <filename>[,A]	Writes to disk the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)
LOAD <filename>[,R]	Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files. (LOAD <filename>,R and RUN <filename>,R are equivalent.)

RUN <filename>[,R]	RUN <filename> loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open. (RUN <filename>,R and LOAD <filename>,R are equivalent.)
MERGE <filename>	Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.
KILL <filename>	Deletes the file from the disk. <filename> may be a program file, or a sequential or random access data file.
NAME <old filename> AS<new filename>	To change the name of a disk file, execute the NAME statement, NAME <oldfile> AS <newfile>. NAME may be used with program files, random files, or sequential files.

4.2 PROTECTED FILE

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

4.3 DISK DATA FILES - SEQUENTIAL AND RANDOM I/O

There are two types of disk data files that may be created and accessed by a BASIC program: sequential files and random access files.

4.3.1 Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is a series of ASCII characters stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with sequential files are:

OPEN	PRINT#	INPUT#	WRITE#
	PRINT# USING	LINE INPUT#	
CLOSE	EOF	LOC	

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode.	OPEN "O",#1,"DATA"
2. Write data to the file using the PRINT# statement. (WRITE# may be used instead.)	PRINT#1,A$;B$;C$
3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode.	CLOSE #1 OPEN "I",#1,"DATA"
4. Use the INPUT# statement to read data from the sequential file into the program.	INPUT#1,X$,Y$,Z$

Program 1 is a short program that creates a sequential file, "DATA", from information you input at the terminal.

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

PROGRAM 1 - CREATE A SEQUENTIAL DATA FILE

Now look at Program 2. It accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1978.

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

PROGRAM 2 - ACCESSING A SEQUENTIAL FILE

Program 2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed. A sector is a 128-byte block of data.

4.3.1.1 Adding Data To A Sequential File -

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. The following procedure can be used to add data to an existing file called "NAMES".

1. OPEN "NAMES" in "I" mode.

2. OPEN a second file called "COPY" in "O" mode.

3. Read in the data in "NAMES" and write it to "COPY".

4. CLOSE "NAMES" and KILL it.

5. Write the new information to "COPY".

6. Rename "COPY" as "NAMES" and CLOSE.

7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program 3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# will read in characters from the disk until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

PROGRAM 3 - ADDING DATA TO A SEQUENTIAL FILE

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

4.3.2 Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk -- it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

OPEN	FIELD	LSET/RSET	GET
PUT	CLOSE	LOC	
MKI$	CVI		
MKS$	CVS		
MKD$	CVD		

4.3.2.1 Creating A Random File -

The following program steps are required to create a random file.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.	`OPEN "R",#1,"FILE",32`
2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.	`FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$`

3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.	`LSET N$=X$`
	`LSET A$=MKS$(AMT)`
	`LSET P$=TEL$`

4. Write the data from the buffer to the disk using the PUT statement.	`PUT #1,CODE%`

Look at Program 4.
It takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

NOTE

Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

PROGRAM 4 - CREATE A RANDOM FILE

4.3.2.2 Accessing A Random File -

The following program steps are required to access a random file:

1. OPEN the file in "R" mode. `OPEN "R",#1,"FILE",32`

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file. `FIELD #1 20 AS N$, 4 AS A$, 8 AS P$`

NOTE

In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer. `GET #1,CODE%`

4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values. `PRINT N$` `PRINT CVS(A$)`

Program 5 accesses the random file "FILE" that was created in Program 4. By inputting the three-digit code at the terminal, the information associated with that code is read from the file and displayed.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

PROGRAM 5 - ACCESS A RANDOM FILE

The LOC function, with random files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file#1 is higher than 50.

Program 6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
        "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:
        IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;" IN STOCK":GOTO 600
630 Q%=Q%-S%
```

```
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
        " REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";
         CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER":
        GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

PROGRAM 6 - INVENTORY

CHAPTER 5

BASIC ASSEMBLY LANGUAGE SUBROUTINES

All versions of Microsoft BASIC have provisions for interfacing with assemblv language subroutines via the USR function and the CALL statement.

The USR function allows assembly language subroutines to be called in the same way BASIC Intrinsic functions are called.

5.1 MEMORY ALLOCATION

IMPORTANT

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s) with the /M: switch.

BASIC uses all memory available from its starting location upwards, so only the topmost locations in memory can be set aside for user subroutines.

If, when an assembly language subroutine is called, more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

The assembly language subroutine may be loaded into memory by means of the operating system, the BASIC POKE statement, or (if the user has the Utility Software Package) routines may be assembled with the MACRO assembler and loaded using the LINK linking loader.

5.2 USR FUNCTION CALLS

The format of the USR function is

USR[<digit>](argument)

where <digit> is from 0 to 9 and the argument is any numeric or string expression. <digit> specifies which USR routine is being called, and corresponds with the digit supplied in the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. The address given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the type of argument that was given. The value in A may be one of the following:

Value in A	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Single precision floating point number
8	Double precision floating point number

If the argument is a number, the [H,L] register pair points to the Floating Point Accumulator (FAC) where the argument is stored.

If the argument is an integer:

FAC-3 contains the lower 8 bits of the argument and
FAC-2 contains the upper 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-3 contains the lowest 8 bits of mantissa and
FAC-2 contains the middle 8 bits of mantissa and
FAC-1 contains the highest 7 bits of mantissa
with leading 1 suppressed (implied). Bit 7 is
the sign of the number (0=positive, 1=negative).
FAC is the exponent minus 128, and the binary
point is to the left of the most significant
bit of the mantissa.

If the argument is a double precision floating point number:

FAC-7 through FAC-4 contain four more bytes
of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the [D,E] register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to

255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

CAUTION: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. Example:

```
A$ = "BASIC"+""
```

This will copy the string literal into string space and will prevent alteration of program text during a subroutine call.

Usually, the value returned by a USR function is the same type (integer, string, single precision or double precision) as the argument that was passed to it. However, calling the MAKINT routine returns the integer in [H,L] as the value of the function, forcing the value returned by the function to be integer. To execute MAKINT, use the following sequence to return from the subroutine:

```
   PUSH    H       ;save value to be returned
   LHLD    xxx     ;get address of MAKINT routine
   XTHL            ;save return on stack and
                   ;get back [H,L]
   RET             ;return
```

Also, the argument of the function, regardless of its type, may be forced to an integer by calling the FRCINT routine to get the integer value of the argument in [H,L]. Execute the following routine:

```
   LXI     H       ;get address of subroutine
                   ;continuation
   PUSH    H       ;place on stack
   LHLD    xxx     ;get address of FRCINT
   PCHL
SUB1: . . . . .
```

5.3 CALL STATEMENT

User function calls may also be made with the CALL statement.

A CALL statement with no arguments generates a simple "CALL" instruction. The corresponding subroutine should return via a simple "RET." (CALL and RET are 8080 opcodes - see an 8080 reference manual for details.)

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL

argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing the parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).

2. If the number of parameters is greater than 3, they are passed as follows:

    1. Parameter 1 in HL.

    2. Parameter 2 in DE.

    3. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that, with this scheme, the subroutine must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for the correct number or tvpe of parameters.

If the subroutine expects more than 3 parameters, and needs to transfer them to a local data area, there is a system subroutine which will perform this transfer. This argument transfer routine is named $AT (located in the FORTRAN library, FORLIB.REL), and is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2). The subroutine is responsible for saving the first two parameters before calling $AT. For example, if a subroutine expects 5 parameters, it should look like:

```
SUBR:       SHLD       P1          ;SAVE PARAMETER 1
            XCHG
            SHLD       P2          ;SAVE PARAMETER 2
            MVI        A,3         ;NO. OF PARAMETERS LEFT
            LXI        H,P3        ;POINTER TO LOCAL AREA
            CALL       $AT         ;TRANSFER THE OTHER 3
PARAMETERS
            .
            .
            .
            [Body of subroutine]
            .
            .
            .
            RET                    ;RETURN TO CALLER
P1:         DS         2           ;SPACE FOR PARAMETER 1
P2:         DS         2           ;SPACE FOR PARAMETER 2
P3:         DS         6           ;SPACE FOR PARAMETERS 3-5
```

A listing of the argument transfer routine $AT follows.

```
;           ARGUMENT TRANSFER
;[B,C]      POINTS TO 3RD PARAM.
;[H,L]      POINTS TO LOCAL STORAGE FOR PARAM 3
;[A]        CONTAINS THE # OF PARAMS TO XFER (TOTAL-2)


            ENTRY   $AT
$AT:        XCHG                ;SAVE [H,L] IN [D,E]
            MOV     H,B
            MOV     L,C         ;[H,L] = PTR TO PARAMS
AT1:        MOV     C,M
            INX     H
            MOV     B,M
            INX     H           ;[B,C] = PARAM ADR
            XCHG                ;[H,L] POINTS TO LOCAL STORAGE
            MOV     M,C
            INX     H
            MOV     M,B
            INX     H           ;STORE PARAM IN LOCAL AREA
            XCHG                ;SINCE GOING BACK TO AT1
            DCR     A           ;TRANSFERRED ALL PARAMS?
            JNZ     AT1         ;NO, COPY MORE
            RET                 ;YES, RETURN
```

When accessing parameters in a subroutine, remember that they are pointers to the actual arguments passed.

NOTE

> The programmer must match the number, type, and length of the arguments in the calling program with the parameters expected by the subroutine . This applies to BASIC subroutines, as well as those written in assembly language.

5.4 INTERRUPTS

Assembly language subroutines can be written to handle interrupts. All interrupt handling routines should save the stack, register AL and the PSW. Interrupts should always be re-enabled before returning from the subroutine because an interrupt automatically disables all further interrupts once it is received. The user should be aware of which interrupt vectors are free in the particular version of BASIC that has been supplied. (Note to CP/M users: In CP/M BASIC, all interrupt vectors are free.)

CHAPTER 6

MICROSOFT BASIC WITH THE CP/M OPERATING SYSTEM

The CP/M version of Microsoft BASIC (MBASIC) is supplied on a standard size 3740 single density diskette. The name of the file is MBASIC.COM. (A 28K or larger CP/M system is recommended.)

To run MBASIC, bring up CP/M and type the following:

```
A>MBASIC <carriage return>
```

The system will reply:

```
xxxx Bytes Free
Microsoft BASIC Version 5.0
(CP/M Version)
Copyright 1978 (C) by Microsoft
Created: dd-mmm-yy
Ok
```

MBASIC is the same as Microsoft BASIC as described in the Microsoft BASIC Reference Manual, with the following exceptions:

6.1 INITIALIZATION

The initialization dialog has been replaced by a set of options which are placed after the MBASIC command to CP/M. The format of the command line is:

```
A>MBASIC [<filename>][/F:<number of files>][/M:<highest memory location>]
      [/S:<maximum record size>]
```

If <filename> is present, MBASIC proceeds as if a RUN <filename> command were typed after initialization is complete. A default extension of .BAS is used if none is supplied and the filename is less than 9 characters long. This allows BASIC programs to be executed in batch mode using the SUBMIT facility of CP/M. Such programs should include a SYSTEM statement (see below) to return to CP/M

when they have finished, allowing the next program in the batch stream to execute.

If /F:<number of files> is present, it sets the number of disk data files that may be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes of memory. If the /F option is omitted, the number of files defaults to 3.

The /M:<highest memory location> option sets the highest memory location that will be used by MBASIC. In some cases it is desirable to set the amount of memory well below the CP/M's FDOS to reserve space for assembly language subroutines. In all cases, <highest memory location> should be below the start of FDOS (whose address is contained in locations 6 and 7). If the /M option is omitted, all memory up to the start of FDOS is used.

/S:<maximum record size> may be added at the end of the command line to set the maximum record size for use with random files. The default record size is 128 bytes.

NOTE

<number of files>, <highest memory location>, and <maximum record size> are numbers that may be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

Examples:

A>MBASIC PAYROLL.BAS	Use all memory and 3 files, load and execute PAYROLL.BAS.
A>MBASIC INVENT/F:6	Use all memory and 6 files, load and execute INVENT.BAS.
A>MBASIC /M:32768	Use first 32K of memory and 3 files.
A>MBASIC DATACK/F:2/M:&H9000	Use first 36K of memory, 2 files, and execute DATACK.BAS.

6.2 DISK FILES

Disk filenames follow the normal CP/M naming conventions. All filenames may include A: or B: as the first two characters to specify a disk drive, otherwise the currently selected drive is assumed. A default extension of .BAS is used on LOAD, SAVE, MERGE and RUN <filename> commands if no "." appears in the filename and the filename is less than 9 characters long.

For systems with CP/M 2.x, large random files are supported. The maximum logical record number is 32767. If a record size of 256 is specified, then files up to 8 megabytes can be accessed.

6.3 FILES COMMAND

Format: FILES[<filename>]

Purpose: To print the names of files residing on the current disk.

Remarks: If <filename> is omitted, all the files on the currently selected drive will be listed. <filename> is a string formula which may contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the filename or extension will match any file or any extension.

Examples:

```
FILES
FILES "*.BAS"
FILES "B:*.*"
FILES "TEST?.BAS"
```

6.4 RESET COMMAND

Format: RESET

Purpose: To close all disk files and write the directory information to a diskette before it is removed from a disk drive.

Remarks: Always execute a RESET command before removing a diskette from a disk drive. Otherwise, when the diskette is used again, it will not have the current directory information written on the directory track.

RESET closes all open files on all drives and writes the directory track to every diskette with open files.

6.5 LOF FUNCTION

Format: LOF(<file number>)

Action: Returns the number of records present in the last extent read or written. If the file does not exceed one extent (128 records), then LOF returns the true length of the file.

Example: 110 IF NUM%>LOF(1) THEN PRINT "INVALID ENTRY"

6.6 EOF

With CP/M, the EOF function may be used with random files. If a GET is done past the end of file, EOF will return -1. This may be used to find the size of a file using a binary search or other algorithm.

6.7 MISCELLANEOUS

1. CSAVE and CLOAD are not implemented.

2. To return to CP/M, use the SYSTEM command or statement. SYSTEM closes all files and then performs a CP/M warm start. Control-C always returns to MBASIC, not to CP/M.

3. FRCINT is at 103 hex and MAKINT is at 105 hex. (Add 1000 hex for ADDS versions, 4000 for SBC CP/M versions.)

INDEX

basic-80 reference manual

This manual is a reference for Microsoft's BASIC-80 language, release 5.0 and later.

There are significant differences between the 5.0 release of BASIC-80 and the previous releases (release 4.51 and earlier). If you have programs written under a previous release of BASIC-80, check Appendix A for new features in 5.0 that may affect execution.

LIMITED WARRANTY

MICROSOFT shall have no liability or responsibility to purchaser or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.

8101-530-08
30F14A

BASIC-80 Reference Manual

CONTENTS

[illegible]

Introduction

BASIC-80 is the most extensive implementation of BASIC available for the 8080 and Z80 microprocessors. In its fifth major release (Release 5.0), BASIC-80 meets the ANSI qualifications for BASIC, as set forth in document BSRX3.60-1978. Each release of BASIC-80 consists of three upward compatible versions: 8K, Extended and Disk . This manual is a reference for all three versions of BASIC-80, release 5.0 and later. This manual is also a reference for Microsoft BASIC-86. BASIC-86 is currently available in Extended and Disk Standalone versions, which are comparable to the BASIC-80 Extended and Disk Standalone versions.

There are significant differences between the 5.0 release of BASIC-80 and the previous releases (release 4.51 and earlier). If you have programs written under a previous release of BASIC-80, check Appendix A for new features in 5.0 that may affect execution.

The manual is divided into three large chapters plus a number of appendices. Chapter 1 covers a variety of topics, largely pertaining to information representation when using BASIC-80. Chapter 2 contains the syntax and semantics of every command and statement in BASIC-80, ordered alphabetically. Chapter 3 describes all of BASIC-80's intrinsic functions, also ordered alphabetically. The appendices contain information pertaining to individual operating systems; plus lists of error messages, ASCII codes, and math functions; and helpful information on assembly language subroutines and disk I/O.

[illegible]

[illegible]

[illegible]

[illegible]

CHAPTER 1

GENERAL INFORMATION ABOUT BASIC-80

1.1 INITIALIZATION

The procedure for initialization will vary with different implementations of BASIC-80. Check the appropriate appendix at the back of this manual to determine how BASIC-80 is initialized with your operating system.

1.2 MODES OF OPERATION

When BASIC-80 is initialized, it types the prompt "Ok". "Ok" means BASIC-80 is at command level, that is, it is ready to accept commands. At this point, BASIC-80 may be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, BASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program.

The indirect mode is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

1.3 LINE FORMAT

Program lines in a BASIC program have the following format (square brackets indicate optional):

```
nnnnn BASIC statement[:BASIC statement...] <carriage return>
```

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of:

72 characters in 8K BASIC-80
255 characters in Extended and Disk BASIC-80.

In Extended and Disk versions, it is possible to extend a logical line over more than one physical line by use of the terminal's <line feed> key. <Line feed> lets you continue typing a logical line on the next physical line without entering a <carriage return>. (In the 8K version, <line feed> has no effect.)

1.3.1 Line Numbers

Every BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 65529. In the Extended and Disk versions, a period (.) may be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

1.4 CHARACTER SET

The BASIC-80 character set is comprised of alphabetic characters, numeric characters and special characters.

The alphabetic characters in BASIC-80 are the upper case and lower case letters of the alphabet.

The numeric characters in BASIC-80 are the digits 0 through 9.

The following special characters and terminal keys are recognized by BASIC-80:

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
<rubout>	Deletes last character typed.
<escape>	Escapes Edit Mode subcommands. See Section 2.16.
<tab>	Moves print position to next tab stop. Tab stops are every eight columns.
<line feed>	Moves to next physical line.
<carriage return>	Terminates input of a line.

1.4.1 Control Characters

The following control characters are in BASIC-80:

Control-A	Enters Edit Mode on the line being typed.
Control-C	Interrupts program execution and returns to BASIC-80 command level.
Control-G	Rings the bell at the terminal.
Control-H	Backspace. Deletes the last character typed.
Control-I	Tab. Tab stops are every eight columns.
Control-O	Halts program output while execution continues. A second Control-O restarts output.
Control-R	Retypes the line that is currently being typed.
Control-S	Suspends program execution.
Control-Q	Resumes program execution after a Control-S.
Control-U	Deletes the line that is currently being typed.

1.5 CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

1. Integer constants — Whole numbers between -32768 and +32767. Integer constants do not have decimal points.

2. Fixed Point constants — Positive or negative real numbers, i.e., numbers that contain decimal points.

3. Floating Point constants — Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10-38 to 10+38.
   Examples:

   235.988E-7 = .0000235988
   2359E6 = 2359000000

   (Double precision floating point constants use the letter D instead of E. See Section 1.5.1.)

4. Hex constants — Hexadecimal numbers with the prefix &H. Examples:

   &H76
   &H32F

5. Octal constants — Octal numbers with the prefix &O or &. Examples:

   &O347
   &1234

1.5.1 Single And Double Precision Form For Numeric Constants

In the 8K version of BASIC-80, all numeric constants are single precision numbers. They are stored with 7 digits of precision, and printed with up to 6 digits.

In the Extended and Disk versions, however, numeric constants may be either single precision or double precision numbers. With double precision, the numbers are stored with 16 digits of precision, and printed with up to 16 digits.

A single precision constant is any numeric constant that has:

1. seven or fewer digits, or
2. exponential form using E, or
3. a trailing exclamation point (!)

A double precision constant is any numeric constant that has:

1. eight or more digits, or
2. exponential form using D, or
3. a trailing number sign (#)

Examples:

Single Precision Constants	Double Precision Constants
46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

1.6 VARIABLES

Variables are names used to represent values that are used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

1.6.1 Variable Names And Declaration Characters

BASIC-80 variable names may be any length, however, in the 8K version, only the first two characters are significant. In the Extended and Disk versions, up to 40 characters are significant. The characters allowed in a variable name are letters and numbers, and the decimal point is allowed in Extended and Disk variable names. The first character must be a letter. Special type declaration characters are also allowed -- see below.

A variable name may not be a reserved word. The Extended and Disk versions allow embedded reserved words; the 8K version does not. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all BASIC-80 commands, statements, function

names and operator names.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign ($) as the last character. For example: A$ = "SALES REPORT". The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string.

In the Extended and Disk versions, numeric variable names may declare integer, single or double precision values. (All numeric values in 8K are single precision.) The type declaration characters for these variable names are as follows:

%	Integer variable
!	Single precision variable
#	Double precision variable

The default type for a numeric variable name is single precision.

Examples of BASIC-80 variable names follow.

In Extended and Disk versions:

PI#	declares a double precision value
MINIMUM!	declares a single precision value
LIMIT%	declares an integer value

In 8K, Extended and Disk versions:

N$	declares a string value
ABC	represents a single precision value

In the Extended and Disk versions of BASIC-80, there is a second method by which variable types may be declared. The BASIC-80 statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Section 2.12.

1.6.2 Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an

array is 255. The maximum number of elements per dimension is 32767.

1.6.3 Space Requirements

VARIABLES: BYTES

INTEGER	2
SINGLE PRECISION	4
DOUBLE PRECISION	8

ARRAYS: BYTES

INTEGER	2 per element
SINGLE PRECISION	4 per element
DOUBLE PRECISION	8 per element

STRINGS:

3 bytes overhead plus the present contents of the string.

1.7 TYPE CONVERSION

When necessary, BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)
   Example:

   ```
   10 A% = 23.42
   20 PRINT A%
   RUN
    23
   ```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.
   Examples:

   ```
   10 D# = 6#/7     The arithmetic was performed
   ```

```
20 PRINT D#       in double precision and the
RUN               result was returned in D#
 .8571428571428571 as a double precision value.

10 D = 6#/7       The arithmetic was performed
20 PRINT D        in double precision and the
RUN               result was returned to D (single
 .857143          precision variable), rounded and
                  printed as a single precision
                  value.
```

3. Logical operators (see Section 1.8.3) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.

4. When a floating point value is converted to an integer, the fractional portion is rounded. Example:

```
10 C% = 55.88
20 PRINT C%
RUN
 56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits, rounded, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8 times the original single precision value. Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.04  2.039999961853027
```

1.8 EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC-80 may be divided into four categories:

1. Arithmetic

2. Relational

3. Logical

4. Functional

1.8.1 Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, Floating Point Division	X*Y X/Y
+,-	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression
$X+2Y$	X+Y*2
$X-\frac{Y}{Z}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
X^{Y^Z}	X^(Y^Z)
$X(-Y)$	X*(-Y) Two consecutive operators must be separated by parentheses.

1.8.1.1 Integer Division And Modulus Arithmetic -

Two additional operators are available in Extended and Disk versions of BASIC-80: Integer division and modulus arithmetic.

Integer division is denoted by the baskslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer.
For example:

```
10\4 = 2
25.68\6.99 = 3
```

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2 (10/4=2 with a remainder 2)
25.68 MOD 6.99 = 5 (26/7=3 with a remainder 5)
```

The precedence of modulus arithmetic is just after integer division.

1.8.1.2 Overflow And Division By Zero -

If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

1.8.2 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then used to make a decision regarding program flow. (See IF, Section 2.26.)

Operator	Relation Tested	Expression
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See LET, Section 2.30.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y < (T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples:

```
IF SIN(X)<0 GOTO 1000
IF I MOD J <> 0 THEN K=K+1
```

1.8.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF, Section 2.26). For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is

performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
-1 AND 8=8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

1.8.4 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC-80 has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC-80's intrinsic functions are described in Chapter 3.

BASIC-80 also allows "user defined" functions that are written by the programmer. See DEF FN, Section 2.11.

1.8.5 String Operations

Strings may be concatenated using +. For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=    <>    <    >    <=    >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"     where B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

1.9 INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the RUBOUT key or with Control-H. Rubout surrounds the deleted character(s) with backslashes, and Control-H has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type Control-U. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. BASIC-80 will automatically replace the old line with the new line.

More sophisticated editing capabilities are provided in the Extended and Disk versions of BASIC-80. See EDIT, Section 2.16.

To delete the entire program that is currently residing in memory, enter the NEW command. (See Section 2.41.) NEW is usually used to clear memory prior to entering a new program.

1.10 ERROR MESSAGES

If BASIC-80 detects an error that causes program execution to terminate, an error message is printed. In the 8K version, only the error code is printed. In the Extended and Disk versions, the entire error message is printed. For a complete list of BASIC-80 error codes and error messages, see Appendix J.

CHAPTER 2

BASIC-80 COMMANDS AND STATEMENTS

All of the BASIC-80 commands and statements are described in this chapter. Each description is formatted as follows:

Format:	Shows the correct format for the instruction. See below for format notation.
Versions:	Lists the versions of BASIC-80 in which the instruction is available.
Purpose:	Tells what the instruction is used for.
Remarks:	Describes in detail how the instruction is used.
Example:	Shows sample programs or program segments that demonstrate the use of the instruction.

<u>Format Notation</u>
Wherever the format for a statement or command is given, the following rules apply:

1. Items in capital letters must be input as shown.

2. Items in lower case letters enclosed in angle brackets (< >) are to be supplied by the user.

3. Items in square brackets ([ ]) are optional.

4. All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.

5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).

2.1 AUTO

Format: AUTO [<line number>[,<increment>]]

Versions: Extended, Disk

Purpose: To generate a line number automatically after every carriage return.

Remarks: AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing Control-C. The line in which Control-C is typed is not saved. After Control-C is typed, BASIC returns to command level.

Example:

AUTO 100,50	Generates line numbers 100, 150, 200 ...
AUTO	Generates line numbers 10, 20, 30, 40 ...

2.2 CALL

Format: CALL <variable name>[(<argument list>)]

Version: Extended, Disk

Purpose: To call an assembly language subroutine.

Remarks: The CALL statement is one way to transfer program flow to an external subroutine. (See also the USR function, Section 3.40)

<variable name> contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name. <argument list> contains the arguments that are passed to the external subroutine. <argument list> may contain only variables.

The CALL statement generates the same calling sequence used by Microsoft's FORTRAN, COBOL and BASIC compilers.

Example:

```
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
          .
          .
          .
```

2.3 CHAIN

Format: CHAIN [MERGE] <filename>[,[<line number exp>] [,ALL][,DELETE<range>]]

Version: Disk

Purpose: To call a program and pass variables to it from the current program.

Remarks: <filename> is the name of the program that is called. Example:

```
CHAIN"PROG1"
```

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. Example:

```
CHAIN"PROG1",1000
```

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed. See Section 2.7. Example:

```
CHAIN"PROG1",1000,ALL
```

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd. Example:

```
CHAIN MERGE"OVRLAY",1000
```

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option. Example:

```
CHAIN MERGE"OVRLAY2",1000,DELETE 1000-5000
```

The line numbers in <range> are affected by the RENUM command.

NOTE: The CHAIN statement with MERGE option leaves the files open and preserves the current OPTION BASE setting.

NOTE: If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables must be restated in the chained program.

2.4 CLEAR

Format: CLEAR [,[<expression1>][,<expression2>]]

Versions: 8K, Extended, Disk

Purpose: To set all numeric variables to zero, all string variables to null, and to close all open files; and, optionally, to set the end of memory and the amount of stack space.

Remarks: <expression1> is a memory location which, if specified, sets the highest location available for use by BASIC-80.

<expression2> sets aside stack space for BASIC. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

NOTE: In previous versions of BASIC-80, <expression1> set the amount of string space, and <expression2> set the end of memory. BASIC-80, release 5.0 and later, allocates string space dynamically. An "Out of string space error" occurs only if there is no free memory left for BASIC to use.

2.5 CLOAD

Formats: CLOAD <filename>

CLOAD? <filename>

CLOAD* <array name>

Versions: 8K (cassette), Extended (cassette)

Purpose: To load a program or an array from cassette tape into memory.

Remarks: CLOAD executes a NEW command before it loads the program from cassette tape. <filename> is the string expression or the first character of the string expression that was specified when the program was CSAVEd.

CLOAD? verifies tapes by comparing the program currently in memory with the file on tape that has the same filename. If they are the same, BASIC-80 prints Ok. If not, BASIC-80 prints NO GOOD.

CLOAD* loads a numeric array that has been saved on tape. The data on tape is loaded into the array called <array name> specified when the array was CSAVE*ed.

CLOAD and CLOAD? are always entered at command level as direct mode commands. CLOAD* may be entered at command level or used as a program statement. Make sure the array has been DIMensioned before it is loaded. BASIC-80 always returns to command level after a CLOAD, CLOAD? or CLOAD* is executed. Before a CLOAD is executed, make sure the cassette recorder is properly connected and in the Play mode, and the tape is possitioned correctly.

See also CSAVE, Section 2.9.

NOTE: CLOAD and CSAVE are not included in all implementations of BASIC-80.

Example: CLOAD "MAX2"

Loads file "M" into memory.

2.6 CLOSE

Format: CLOSE[[#]<file number>[,[#]<file number...>]]

Version: Disk

Purpose: To conclude I/O to a disk file.

Remarks: <file number> is the number under which the file was OPENed. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disk files automatically. (STOP does not close disk files.)

Example: See Appendix B.

2.7 COMMON

Format: COMMON <list of variables>

Version: Disk

Purpose: To pass variables to a CHAINed program.

Remarks: The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:

```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
        .
        .
        .
```

2.8 CONT

Format: CONT

Versions: 8K, Extended, Disk

Purpose: To continue program execution after a Control-C has been typed, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (? or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. With the Extended and Disk versions, CONT may be used to continue execution after an error.

CONT is invalid if the program has been edited during the break. In 8K BASIC-80, execution cannot be CONTinued if a direct mode error has occurred during the break.

Example: See example Section 2.61, STOP.

2.9 CSAVE

Formats: CSAVE <string expression>

CSAVE* <array variable name>

Versions: 8K (cassette), Extended (cassette)

Purpose: To save the program or an array currently in memory on cassette tape.

Remarks: Each program or array saved on tape is identified by a filename. When the command CSAVE <string expression> is executed, BASIC-80 saves the program currently in memory on tape and uses the first character in <string expression> as the filename. <string expression> may be more than one character, but only the first character is used for the filename.

When the command CSAVE* <array variable name> is executed, BASIC-80 saves the specified array on tape. The array must be a numeric array. The elements of a multidimensional array are saved with the leftmost subscript changing fastest.

CSAVE may be used as a program statement or as a direct mode command.

Before a CSAVE or CSAVE* is executed, make sure the cassette recorder is properly connected and in the Record mode.

See also CLOAD, Section 2.5.

NOTE: CSAVE and CLOAD are not included in all implementations of BASIC-80.

Example: CSAVE "TIMER"

Saves the program currently in memory on cassette under filename "T".

2.10 DATA

Format: DATA <list of constants>

Versions: 8K, Extended, Disk

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s). (See READ, Section 2.54)

Remarks: DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement (Section 2.57).

Example: See examples in Section 2.54, READ.

2.11 DEF FN

Format: DEF FN<name>[(<parameter list>)]=<function definition>

Versions: 8K, Extended, Disk

Purpose: To define and name a function that is written by the user.

Remarks: <name> must be a legal variable name. This name, preceded by FN, becomes the name of the function. <parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. <function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call. (Remember, in the 8K version only one argument is allowed in a function call, therefore the DEF FN statement will contain only one variable.)

In Extended and Disk BASIC-80, user-defined functions may be numeric or string; in 8K, user-defined string functions are not allowed. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example:

```
       .
       .
410 DEF FNAB(X,Y)=X^3/Y^2
420 T=FNAB(I,J)
       .
       .
```

Line 410 defines the function FNAB. The function is called in line 420.

2.12 DEFINT/SNG/DBL/STR

Format: DEF<type> <range(s) of letters>
where <type> is INT, SNG, DBL, or STR

Versions: Extended, Disk

Purpose: To declare variable types as integer, single precision, double precision, or string.

Remarks: A DEFtype statement declares that the variable names beginning with the letter(s) specified will be that type variable. However, a type declaration character always takes precedence over a DEFtype statement in the typing of a variable.

If no type declaration statements are encountered, BASIC-80 assumes all variables without declaration characters are single precision variables.

Examples:

```
10 DEFDBL L-P    All variables beginning with
                 the letters L, M, N, O, and P
                 will be double precision
                 variables.

10 DEFSTR A      All variables beginning with
                 the letter A will be string
                 variables.

10 DEFINT I-N,W-Z
                 All variable beginning with
                 the letters I, J, K, L, M,
                 N, W, X, Y, Z will be integer
                 variables.
```

2.13 DEF USR

Format: DEF USR[<digit>]=<integer expression>

Versions: Extended, Disk

Purpose: To specify the starting address of an assembly language subroutine.

Remarks: <digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. See Appendix C, Assembly Language Subroutines.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example:

```
 .
 .
 .
200 DEF USR0=24000
210 X=USR0(Y^2/2.89)
 .
 .
 .
```

2.14 DELETE

Format: DELETE[<line number>][-<line number>]

Versions: Extended, Disk

Purpose: To delete program lines.

Remarks: BASIC-80 always returns to command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

Examples:

DELETE 40	Deletes line 40
DELETE 40-100	Deletes lines 40 through 100, inclusive
DELETE-40	Deletes all lines up to and including line 40

2.15 DIM

Format: DIM <list of subscripted variables>

Versions: 8K, Extended, Disk

Purpose: To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see Section 2.46).

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example:

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
     .
     .
     .
```

2.16 EDIT

Format: EDIT <line number>

Versions: Extended, Disk

Purpose: To enter Edit Mode at the specified line.

Remarks: In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering Edit Mode, BASIC-80 types the line number of the line to be edited, then it types a space and waits for an Edit Mode subcommand.

Edit Mode Subcommands

Edit Mode subcommands are used to move the cursor or to insert, delete, replace, or search for text within a line. The subcommands are not echoed. Most of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When a preceding integer is not specified, it is assumed to be 1.

Edit Mode subcommands may be categorized according to the following functions:

1. Moving the cursor
2. Inserting text
3. Deleting text
4. Finding text
5. Replacing text
6. Ending and restarting Edit Mode

NOTE

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and $ represents the Escape (or Altmode) key.

1. Moving the Cursor

Space — Use the space bar to move the cursor to the right. [i]Space moves the cursor i spaces to the right. Characters are printed as you space over them.

Rubout — In Edit Mode, [i]Rubout moves the cursor i spaces to the left (backspaces). Characters are printed as you backspace over them.

2. Inserting Text

I — I<text>$ inserts <text> at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, type Escape. If Carriage Return is typed during an Insert command, the effect is the same as typing Escape and then Carriage Return. During an Insert command, the Rubout, Delete, or Underscore key on the terminal may be used to delete characters to the left of the cursor. Rubout will print out the characters as you backspace over them. Delete and Underscore will print an Underscore for each character that you backspace over. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (Control-G) is typed and the character is not printed.

X — The X subcommand is used to extend the line. X moves the cursor to the end of the line, goes into insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, type Escape or Carriage Return.

3. Deleting Text

D — [i]D deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, iD deletes the remainder of the line.

H — H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for replacing statements at the end of a line.

4. Finding Text

S — The subcommand [i]S<ch> searches for the ith

occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed.

K The subcommand [i]K<ch> is similar to [i]S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

5. Replacing Text

C The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by i characters. After the ith new character is typed, change mode is exited and you will return to Edit Mode.

6. Ending and Restarting Edit Mode

<cr> Typing Carriage Return prints the remainder of the line, saves the changes you made and exits Edit Mode.

E The E subcommand has the same effect as Carriage Return, except the remainder of the line is not printed.

Q The Q subcommand returns to BASIC-80 command level, <u>without</u> saving any of the changes that were made to the line during Edit Mode.

L The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in Edit Mode. L is usually used to list the line when you first enter Edit Mode.

A The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

NOTE

If BASIC-80 receives an unrecognizable command or illegal character while in Edit Mode, it prints a bell (Control-G) and the command or character is ignored.

Syntax Errors

When a Syntax Error is encountered during execution of a program, BASIC-80 automatically enters Edit Mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and type Carriage Return (or the E subcommand), BASIC-80 reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination , first exit Edit Mode with the Q subcommand. BASIC-80 will return to command level, and all variable values will be preserved.

Control-A

To enter Edit Mode on the line you are currently typing, type Control-A. BASIC-80 responds with a carriage return, an exclamation point (!) and a space. The cursor will be positioned at the first character in the line. Proceed by typing an Edit Mode subcommand.

NOTE

Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT." will enter Edit Mode at the current line. (The line number symbol "." always refers to the current line.)

2.17 END

Format: END

Versions: 8K, Extended, Disk

Purpose: To terminate program execution, close all files and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. BASIC-80 always returns to command level after an END is executed.

Example:

```
520 IF K>1000 THEN END ELSE GOTO 20
```

2.18 ERASE

Format: ERASE <list of array variables>

Versions: Extended, Disk

Purpose: To eliminate arrays from a program.

Remarks: Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Redimensioned array" error occurs.

Example:

```
 .
 .
 .
450 ERASE A,B
460 DIM B(99)
 .
 .
 .
```

2.19 ERR AND ERL VARIABLES

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use IF 65535 = ERL THEN ...
Otherwise, use

```
IF ERR = error code THEN ...

IF ERL = line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. BASIC-80's error codes are listed in Appendix J. (For Standalone Disk BASIC error codes, see Appendix H.)

2.20 ERROR

Format: ERROR <integer expression>

Versions: Extended, Disk

Purpose: 1) To simulate the occurrence of a BASIC-80 error; or 2) to allow error codes to be defined by the user.

Remarks: The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC-80 (see Appendix J), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by BASIC-80's error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to BASIC-80.) This user-defined error code may then be conveniently handled in an error trap routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, BASIC-80 responds with the message UNPRINTABLE ERROR. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example 1:

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15              (you type this line)
String too long       (BASIC-80 types this line)
Ok
```

Example 2:

```
 .
 .
 .
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B > 5000 THEN ERROR 210
 .
 .
 .
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL = 130 THEN RESUME 120
 .
 .
 .
```

2.21 FIELD

Format: FIELD[#]<file number>,<field width> AS <string variable>.

Version: Disk

Purpose: To allocate space for variables in a random file buffer.

Remarks: To get data out of a random buffer after a GET or to enter data before a PUT, a FIELD statement must have been executed.

<file number> is the number under which the file was OPENed. <field width> is the number of characters to be allocated to <string variable>. For example,

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See LSET/RSET and GET.)

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128.)

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

Example: See Appendix B.

NOTE: Do not use a FIELDed variable name in an INPUT or LET statement. Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

2.22 FOR...NEXT

Format: FOR <variable>=x TO y [STEP z]

.
.
.

NEXT [<variable>][,<variable>...]

where x, y and z are numeric expressions.

Versions: 8K, Extended, Disk

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: <variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, BASIC-80 branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be

omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1:

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
Ok
```

Example 2:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3:

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set. (Note: Previous versions of BASIC-80 set the initial value of the loop variable before setting the final value; i.e., the above loop would have executed six times.)

2.23 GET

Format: GET [#]<file number>[,<record number>]

Version: Disk

Purpose: To read a record from a random disk file into a random buffer.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

Example: See Appendix B.

NOTE: After a GET statement, INPUT# and LINE INPUT# may be done to read characters from the random file buffer.

2.24 GOSUB...RETURN

Format: GOSUB <line number>

```
GOSUB <line number>
  .
  .
  .
RETURN
```

Versions: 8K, Extended, Disk

Purpose: To branch to and return from a subroutine.

Remarks: <line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC-80 to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertant entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

2.25 GOTO

Format: GOTO <line number>

Versions: 8K, Extended, Disk

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example:

```
LIST
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5          AREA = 78.5
R = 7          AREA = 153.86
R = 12         AREA = 452.16
?Out of data in 10
Ok
```

2.26 IF...THEN[...ELSE] AND IF...GOTO

Format: IF <expression> THEN <statement(s)> | <line number>

[ELSE <statement(s)> | <line number>]

Format: IF <expression> GOTO <line number>

[ELSE <statement(s)> | <line number>]

Versions: 8K, Extended, Disk

NOTE: The ELSE clause is allowed only in Extended and Disk versions.

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. (ELSE is allowed only in Extended and Disk versions.) Extended and Disk versions allow a comma before THEN.

Nesting of IF Statements

In the Extended and Disk versions, IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
     THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example

```
IF A=B THEN IF B=C THEN PRINT "A=C"
     FLSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in the indirect mode.

NOTE: When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1:

```
200 IF I THEN GET#1,I
```

This statement GETs record number I if I is not zero.

Example 2:

```
100 IF(I<20)*(I>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
      .
      .
      .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3:

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

2.27 INPUT

Format: INPUT[;][<"prompt string">;]<list of variables>

Versions: 8K, Extended, Disk

Purpose: To allow input from the terminal during program execution.

Remarks: When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not echo a carriage return/line feed sequence.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the messsage "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

In the 8K version, INPUT is illegal in the direct mode.

Examples:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
? 5      (The 5 was typed in by the user
          in response to the question mark.)
 5 SQUARED IS 25
Ok


LIST
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS?  7.4  (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
etc.
```

2.28 INPUT#

Format: INPUT#<file number>,<variable list>

Version: Disk

Purpose: To read data items from a sequential disk file and assign them to program variables.

Remarks: <file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma.

If BASIC-80 is scanning the sequential data file for a string item, leading spaces, carriage returns and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

Example: See Appendix B.

2.29 KILL

Format: KILL <filename>

Version: Disk

Purpose: To delete a file from disk.

Remarks: If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

KILL is used for all types of disk files: program files, random data files and sequential data files.

Example: 200 KILL "DATA1"

See also Appendix B.

2.30 LET

Format: [LET] <variable>=<expression>

Versions: 8K, Extended, Disk

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
          .
          .
          .

    or

110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
          .
          .
          .
```

2.31 LINE INPUT

Format: LINE INPUT[;][<"prompt string">;]<string variable>

Versions: Extended, Disk

Purpose: To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks: The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to <string variable>. However, if a line feed/carriage return sequence (this order only) is encountered, both characters are echoed; but the carriage return is ignored, the line feed is put into STRING variable>, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be escaped by typing Control-C. BASIC-80 will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT.

Example: See Example, Section 2.32, LINE INPUT#.

2.32 LINE INPUT#

Format: LINE INPUT#<file number>,<string variable>

Version: Disk

Purpose: To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks: <file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC-80 program saved in ASCII mode is being read as data by another program.

Example:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES    234,4    MEMPHIS
LINDA JONES    234,4    MEMPHIS
Ok
```

2.33 LIST

Format 1: LIST [<line number>]

Versions: 8K, Extended, Disk

Format 2: LIST [<line number>[-[<line number>]]]

Versions: Extended, Disk

Purpose: To list all or part of the program currently in memory at the terminal.

Remarks: BASIC-80 always returns to command level after a LIST is executed.

Format 1: If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by typing Control-C.) If <line number> is included, the 8K version will list the program beginning at that line; and the Extended and Disk versions will list only the specified line.

Format 2: This format allows the following options:

1. If only the first number is specified, that line and all higher-numbered lines are listed.

2. If only the second number is specified, all lines from the beginning of the program through that line are listed.

3. If both numbers are specified, the entire range is listed.

Examples: Format 1:

LIST	Lists the program currently in memory.
LIST 500	In the 8K version, lists all programs lines from 500 to the end. In Extended and Disk, lists line 500.

Format 2:

LIST 150-	Lists all lines from 150 to the end.
LIST -1000	Lists all lines from the lowest number through 1000.
LIST 150-1000	Lists lines 150 through 1000, inclusive.

2.34 LLIST

Format: LLIST [<line number>[-[<line number>]]]

Versions: Extended, Disk

Purpose: To list all or part of the program currently in memory at the line printer.

Remarks: LLIST assumes a 132-character wide printer.

BASIC-80 always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Format 2.

NOTE: LLIST and LPRINT are not included in all implementations of BASIC-80.

Example: See the examples for LIST, Format 2.

2.35 LOAD

Format: LOAD <filename>[,R]

Version: Disk

Purpose: To load a file from disk into memory.

Remarks: <filename> is the name that was used when the file was SAVEd. (With CP/M, the default extension .BAS is supplied.)

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the "R" option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD with the "R" option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

Example: LOAD "STRTRK",R

2.36 LPRINT AND LPRINT USING

Format: LPRINT [<list of expressions>]

LPRINT USING <string exp>;<list of expressions>

Versions: Extended, Disk

Purpose: To print data at the line printer.

Remarks: Same as PRINT and PRINT USING, except output goes to the line printer. See Section 2.49 and Section 2.50.

LPRINT assumes a 132-character-wide printer.

NOTE: LPRINT and LLIST are not included in all implementations of BASIC-80.

2.37 LSET AND RSET

Format: LSET <string variable> = <string expression>
RSET <string variable> = <string expression>

Version: Disk

Purpose: To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks: If <string expression> requires fewer bytes than were FIELDed to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See the MKI$, MKS$, MKD$ functions, Section 3.25.

Examples:

```
150 LSET A$=MKS$(AMT)
160 LSET D$=DESC($)
```

See also Appendix B.

NOTE: LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

2.38 MERGE

Format: MERGE <filename>

Version: Disk

Purpose: To merge a specified disk file into the program currently in memory.

Remarks: <filename> is the name used when the file was SAVEd. (With CP/M, the default extension .BAS is supplied.) The file must have been SAVEd in ASCII format. (If not, a "Bad file mode" error occurs.)

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disk into the program in memory.)

BASIC-80 always returns to command level after executing a MERGE command.

Example: MERGE "NUMBRS"

2.39 MID$

Format: MID$(<string exp1>,n[,m])=<string exp2>

where n and m are integer expressions and <string exp1> and <string exp2> are string expressions.

Versions: Extended, Disk

Purpose: To replace a portion of one string with another string.

Remarks: The characters in <string exp1>, beginning at position n, are replaced by the characters in <string exp2>. The optional m refers to the number of characters from <string exp2> that will be used in the replacement. If m is omitted, all of <string exp2> is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of <string exp1>.

Example:

```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
RUN
KANSAS CITY, KS
```

MID$ is also a function that returns a substring of a given string. See Section 3.24.

2.40 NAME

Format: NAME <old filename> AS <new filename>

Version: Disk

Purpose: To change the name of a disk file.

Remarks: <old filename> must exist and <new filename> must not exist; otherwise an error will result. After a NAME command, the file exists on the same disk, in the same area of disk space, with the new name.

Example:

```
Ok
NAME "ACCTS" AS "LEDGER"
Ok
```

In this example, the file that was formerly named ACCTS will now be named LEDGER.

2.41 NEW

Format: NEW

Versions: 8K, Extended, Disk

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC-80 always returns to command level after a NEW is executed.

2.42 NULL

Format: NULL <integer expression>

Versions: 8K, Extended, Disk

Purpose: To set the number of nulls to be printed at the end of each line.

Remarks: For 10-character-per-second tape punches, <integer expression> should be >=3. When tapes are not being punched, <integer expression> should be 0 or 1 for Teletypes and Teletype-compatible CRTs. <integer expression> should be 2 or 3 for 30 cps hard copy printers. The default value is 0.

Example:

```
Ok
NULL 2
Ok
100 INPUT X
200 IF X<50 GOTO 800
      .
      .
      .
```

Two null characters will be printed after each line.

2.43 ON ERROR GOTO

Format: ON ERROR GOTO <line number>

Versions: Extended, Disk

Purpose: To enable error trapping and specify the first line of the error handling subroutine.

Remarks: Once error trapping has been enabled all errors detected, including direct mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine. If <line number> does not exist, an "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC-80 to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

NOTE: If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

Example:

```
10 ON ERROR GOTO 1000
```

2.44 ON...GOSUB AND ON...GOTO

Format: ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

Versions: 8K, Extended, Disk

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks: The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

Example:

```
100 ON L-1 GOTO 150,300,320,390
```

2.45 OPEN

Format: OPEN <mode>,[#]<file number>,<filename>,[<reclen>]

Version: Disk

Purpose: To allow I/O to a disk file.

Remarks: A disk file must be OPENed before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose first character is one of the following:

O	specifies sequential output mode
I	specifies sequential input mode
R	specifies random input/output mode

<file number> is an integer expression whose value is between one and fifteen. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.

<filename> is a string expression containing a name that conforms to your operating system's rules for disk filenames.

<reclen> is an integer expression which, if included, sets the record length for random files. The default record length is 128 bytes. See also page A-3.

NOTE: A file can be OPENed for sequential input or random access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

Example:

```
10 OPEN "I",2,"INVEN"
```

See also Appendix B.

2.46 OPTION BASE

Format: OPTION BASE n
where n is 1 or 0

Versions: 8K, Extended, Disk

Purpose: To declare the minimum value for array subscripts.

Remarks: The default base is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is one.

2.47 OUT

Format: OUT I,J
where I and J are integer expressions in the range 0 to 255.

Versions: 8K, Extended, Disk

Purpose: To send a byte to a machine output port.

Remarks: The integer expression I is the port number, and the integer expression J is the data to be transmitted.

Example:

```
100 OUT 32,100
```

2.48 POKE

Format: POKE I,J
where I and J are integer expressions

Versions: 8K, Extended, Disk

Purpose: To write a byte into a memory location.

Remarks: The integer expression I is the address of the memory location to be POKEd. The integer expression J is the data to be POKEd. J must be in the range 0 to 255. In the 8K version, I must be less than 32768. In the Extended and Disk versions, I must be in the range 0 to 65536.

With the 8K version, data may be POKEd into memory locations above 32768 by supplying a negative number for I. The value of I is computed by subtracting 65536 from the desired address. For example, to POKE data into location 45000, I = 45000-65536, or -20536.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Section 3.27.

POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example:

```
10 POKE &H5A00,&HFF
```

2.49 PRINT

Format: PRINT [<list of expressions>]

Versions: 8K, Extended, Disk

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC-80 divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, BASIC-80 goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8(-7) is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-15 is output as .000000000000001 and 1D-16 is output as 1D-16.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1:

```
10 X=5
20 PRINT X+5, X-5, X*(-5), X^5
30 END
RUN
 10            0             -25            3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2:

```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

Example 3:

```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

2.50 PRINT USING

Format: PRINT USING <string exp>;<list of expressions>

Versions: Extended, Disk

Purpose: To print strings or numbers using a specified format.

Remarks and Examples: <list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <string exp> is a string literal (or variable) comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!" Specifies that only the first character in the given string is to be printed.

"\n spaces\" Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is lonnger than the string, the string will be left-justified in the field and padded with spaces on the right. Example:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\  \";A$;B$
50 PRINT USING "\   \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT  !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input. Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

. A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
 0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##   ";10.2,5.3,66.789,.234
10.20   5.30   66.79   0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

\+ A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

- A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##    ";-68.95,2.4,55.6,-.9
-68.95    +2.40    +55.60    -0.90

PRINT USING "##.##-    ";-68.95,22.449,-7.01
68.95-    22.45     7.01-
```

** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.#    ";12.39,-0.9,765.1
*12.4    *-0.9    765.1
```

$$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
 $456.78
```

**$ The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

, A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^^) format.

```
PRINT USING "####,.##";1234.5
1,234.50

PRINT USING "####.##,";1234.5
1234.50,
```

^^^^ Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
 2.35E+02

PRINT USING ".####^^^^-";888888
 .8889E+06

PRINT USING "+.##^^^^";123
+.12E+03
```

_ An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "__" in the format string.

% If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

2.51 PRINT# AND PRINT# USING

Format: PRINT#<filenumber>,[USING<string exp>;]<list of exps>

Version: Disk

Purpose: To write data to a sequential disk file.

Remarks: <file number> is the number used when the file was OPENed for output. <string exp> is comprised of formatting characters as described in Section 2.50, PRINT USING. The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it will be input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement

```
PRINT#1,A$;B$
```

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem. insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;",";B$
```

The image written to disk is

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The statement

```
PRINT#1,A$;B$
```

would write the following image to disk:

```
CAMERA, AUTOMATIC   93604-1
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$. To separate these strings properly on the disk, write double quotes to the disk image using CHR$(34). The statement

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disk:

```
"CAMERA, AUTOMATIC""   93604-1"
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

For more examples using PRINT#, see Appendix B.

See also WRITE#, Section 2.68.

2.52 PUT

Format: PUT [#]<file number>[,<record number>]

Version: Disk

Purpose: To write a record from a random buffer to a random disk file.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the record will have the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

Example: See Appendix B.

NOTE: PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before a PUT statement.

In the case of WRITE#, BASIC-80 pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

2.53 RANDOMIZE

Format: RANDOMIZE [<expression>]

Versions: Extended, Disk

Purpose: To reseed the random number generator.

Remarks: If <expression> is omitted, BASIC-80 suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)? 3  (user
types 3)
 .88598  .484668  .586328  .119426  .709225
Ok
RUN
Random Number Seed (-32768 to 32767)? 4 (user
types 4 for new sequence)
 .803506  .162462  .929364  .292443  .322921
Ok
RUN
Random Number Seed (-32768 to 32767)? 3 (same
sequence as first RUN)
 .88598  .484668  .586328  .119426  .709225
Ok
```

2.54 READ

Format: READ <list of variables>

Versions: 8K, Extended, Disk

Purpose: To read values from a DATA statement and assign them to variables. (See DATA, Section 2.10.)

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statment. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see RESTORE, Section 2.57)

Example 1:

```
.
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2:

```
LIST
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY          STATE          ZIP
DENVER,       COLORADO       80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

2.55 REM

Format: REM <remark>

Versions: 8K, Extended, Disk

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

In the Extended and Disk versions, remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM.

WARNING: Do not use this in a data statement as it would be considered legal data.

Example:

```
     .
     .
     .
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
     .
     .
     .
```

or, with Extended and Disk versions:

```
     .
     .
     .
120 FOR I=1 TO 20     'CALCULATE AVERAGE VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
     .
     .
     .
```

2.56 RENUM

Format: RENUM [[<new number>][,[<old number>][,<increment>]]]

Versions: Extended, Disk

Purpose: To renumber program lines.

Remarks: <new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

NOTE: RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

Examples:

RENUM	Renumbers the entire program. The first new line number will be 10. Lines will increment by 10.
RENUM 300,,50	Renumbers the entire program. The first new line number will be 300. Lines will increment by 50.
RENUM 1000,900,20	Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

2.57 RESTORE

Format: RESTORE [<line number>]

Versions: 8K, Extended, Disk

Purpose: To allow DATA statements to be reread from a specified line.

Remarks: After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
      .
      .
      .
```

2.58 RESUME

Formats: RESUME

RESUME 0

RESUME NEXT

RESUME <line number>

Versions: Extended, Disk

Purpose: To continue program execution after an error recovery procedure has been performed.

Remarks: Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME <line number>	Execution resumes at <line number>.

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

Example:

```
10 ON ERROR GOTO 900
   .
   .
   .
900 IF (ERR=230)AND(ERL=90) THEN PRINT "TRY
AGAIN":RESUME 80
   .
   .
   .
```

2.59 RUN

Format 1: RUN [<line number>]

Versions: 8K, Extended, Disk

Purpose: To execute the program currently in memory.

Remarks: If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC-80 always returns to command level after a RUN is executed.

Example: RUN

Format 2: RUN <filename>[,R]

Version: Disk

Purpose: To load a file from disk into memory and run it.

Remarks: <filename> is the name used when the file was SAVEd. (With CP/M and ISIS-II, the default extension .BAS is supplied.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain OPEN.

Example: RUN "NEWFIL",R

See also Appendix B.

2.60 SAVE

Format: SAVE <filename>[,A | ,P]

Version: Disk

Purpose: To save a program file on disk.

Remarks: <filename> is a quoted string that conforms to your operating system's requirements for filenames. (With CP/M, the default extension .BAS is supplied.) If <filename> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires and ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

Examples:

```
SAVE"COM2",A
SAVE"PROG",P
```

See also Appendix B.

2.61 STOP

Format: STOP

Versions: 8K, Extended, Disk

Purpose: To terminate program execution and return to command level.

Remarks: STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

```
Break in line nnnnn
```

Unlike the END statement, the STOP statement does not close files.

BASIC-80 always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Section 2.8).

Example:

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```

2.62 SWAP

Format: SWAP <variable>,<variable>

Versions: Extended, Disk

Purpose: To exchange the values of two variables.

Remarks: Any type variable may be SWAPped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example:

```
LIST
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
 ONE FOR ALL
 ALL FOR ONE
Ok
```

2.63 TRON/TROFF

Format: TRON

TROFF

Versions: Extended, Disk

Purpose: To trace the execution of program statements.

Remarks: As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example:

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10][20][30][40] 1  10  20
[50][60][30][40] 2  20  30
[50][60][70]
Ok
TROFF
Ok
```

2.64 WAIT

Format: WAIT <port number>, I[,J]
where I and J are integer expressions

Versions: 8K, Extended, Disk

Purpose: To suspend program execution while monitoring the status of a machine input port.

Remarks: The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, BASIC-80 loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero

CAUTION: It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine.

Example: 100 WAIT 32,2

2.65 WHILE...WEND

Format:

```
WHILE <expression>
   .
   .
[<loop statements>]
   .
   .
WEND
```

Versions: Extended, Disk

Purpose: To execute a series of statements in a loop as long as a given condition is true.

Remarks: If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example:

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115       FLIPS=0
120       FOR I=1 TO J-1
130               IF A$(I)>A$(I+1) THEN
                         SWAP A$(I),A$(I+1):FLIPS=1
140       NEXT I
150 WEND
```

2.66 WIDTH

Format: WIDTH [LPRINT] <integer expression>

Versions: Extended, Disk

Purpose: To set the printed line width in number of characters for the terminal or line printer.

Remarks: If the LPRINT option is omitted, the line width is set at the terminal. If LPRINT is included, the line width is set at the line printer.

<integer expression> must have a value in the range 15 to 255. The default width is 72 characters.

If <integer expression> is 255, the line width is "infinite," that is, BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

Example:

```
10 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
WIDTH 18
Ok
RUN
ABCDEFGHIJKLMNOPQR
STUVWXYZ
Ok
```

2.67 WRITE

Format: WRITE[<list of expressions>]

Version: Disk

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of te expressions are output at thee terminal. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement, Section 2.49.

Example:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
 80, 90,"THAT'S ALL"
Ok
```

2.68 WRITE#

Format: WRITE#<file number>,<list of expressions>

Version: Disk

Purpose: To write data to a sequential file.

Remarks: <file number> is the number under which the file was OPENed in "O" mode. The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

Example: Let A$="CAMERA" and B$="93604-1". The statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.

CHAPTER 3

BASIC-80 FUNCTIONS

The intrinsic functions provided by BASIC-80 are presented in this chapter. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

X and Y	Represent any numeric expressions
I and J	Represent integer expressions
X$ and Y$	Represent string expressions

If a floating point value is supplied where an integer is required, BASIC-80 will round the fractional portion and use the resulting integer.

3.1 ABS

Format: ABS(X)

Versions: 8K, Extended, Disk

Action: Returns the absolute value of the expression X.

Example:

```
PRINT ABS(7*(-5))
 35
Ok
```

3.2 ASC

Format: ASC(X$)

Versions: 8K, Extended, Disk

Action: Returns a numerical value that is the ASCII code of the first character of the string X$. (See Appendix M for ASCII codes.) If X$ is null, an "Illegal function call" error is returned.

Example:

```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

See the CHR$ function for ASCII-to-string conversion.

3.3 ATN

Format: ATN(X)

Versions: 8K, Extended, Disk

Action: Returns the arctangent of X in radians. Result is in the range -pi/2 to pi/2. The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example:

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.24905
Ok
```

3.4 CDBL

Format: CDBL(X)

Versions: Extended, Disk

Action: Converts X to a double precision number.

Example:

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67  454.6700134277344
Ok
```

3.5 CHR$

Format: CHR$(I)

Versions: 8K, Extended, Disk

Action: Returns a string whose one element has ASCII code I. (ASCII codes are listed in Appendix M.) CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR$(7)) as a preface to an error message, or a form feed could be sent (CHR$(12)) to clear a CRT screen and return the cursor to the home position.

Example:

```
PRINT CHR$(66)
B
Ok
```

See the ASC function for ASCII-to-numeric conversion.

3.6 CINT

Format: CINT(X)

Versions: Extended, Disk

Action: Converts X to an integer by rounding the fractional portion. If X is not in the range -32768 to 32767, an "Overflow" error occurs.

Example:

```
PRINT CINT(45.67)
 46
Ok
```

See the CDBL and CSNG functions for converting numbers to the double precision and single precision data type. See also the FIX and INT functions, both of which return integers.

3.7 COS

Format: COS(X)

Versions: 8K, Extended, Disk

Action: Returns the cosine of X in radians. The calculation of COS(X) is performed in single precision.

Example:

```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.84212
Ok
```

3.8 CSNG

Format: CSNG(X)

Versions: Extended, Disk

Action: Converts X to a single precision number.

Example:

```
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
RUN
 975.3421  975.342
Ok
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types.

3.9 CVI, CVS, CVD

Format: CVI(<2-byte string>)
CVS(<4-byte string>)
CVD(<8-byte string>)

Version: Disk

Action: Convert string values to numeric values. Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Example:

```
 .
 .
 .
70 FIELD #1,4 AS N$, 12 AS B$, ...
80 GET #1
90 Y=CVS(N$)
 .
 .
 .
```

See also MKI$, MKS$, MKD$, Section 3.25 and Appendix B.

3.10 EOF

Format: EOF(<file number>)

Version: Disk

Action: Returns -1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while INPUTting, to avoid "Input past end" errors.

Example:

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
 .
 .
 .
```

3.11 EXP

Format: EXP(X)

Versions: 8K, Extended, Disk

Action: Returns e to the power of X. X must be <=87.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.5982
Ok
```

3.12 FIX

Format: FIX(X)

Versions: Extended, Disk

Action: Returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

Examples:

```
PRINT FIX(58.75)
 58
Ok

PRINT FIX(-58.75)
-58
Ok
```

3.13 FRE

Format: FRE(0)
FRE(X$)

Versions: 8K, Extended, Disk

Action: Arguments to FRE are dummy arguments. FRE returns the number of bytes in memory not being used by BASIC-80.

FRE("") forces a garbage collection before returning the number of free bytes. BE PATIENT: garbage collection may take 1 to 1-1/2 minutes. BASIC will not initiate garbage collection until all free memory has been used up. Therefore, using FRE("") periodically will result in shorter delays for each garbage collection.

Example:

```
PRINT FRE(0)
 14542
Ok
```

3.14 HEX$

Format: HEX$(X)

Versions: Extended, Disk

Action: Returns a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX$(X) is evaluated.

Example:

```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

See the OCT$ function for octal conversion.

3.15 INKEY$

Format: INKEY$

Action: Returns either a one-character string containing a character read from the terminal or a null string if no character is pending at the terminal. No characters will be echoed and all characters are passed through tto the program except for Control-C, which terminates the program.

Example:

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$=""
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$ : IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0 : RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

3.16 INP

Format: INP(I)

Versions: 8K, Extended, Disk

Action: Returns the byte read from port I. I must be in the range 0 to 255. INP is the complementary function to the OUT statement, Section 2.47.

Example: 100 A=INP(255)

3.17 INPUT$

Format: INPUT$(X[,[#]Y])

Version: Disk

Action: Returns a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters will be echoed and all control characters are passed through except Control-C, which is used to interrupt the execution of the INPUT$ function.

Example 1:

```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN
HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

Example 2:

```
   .
   .
   .
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
   .
   .
   .
```

3.18 INSTR

Format: INSTR([I,]X$,Y$)

Versions: Extended, Disk

Action: Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found. Optional offset I sets the position for starting the search. I must be in the range 1 to 255. If I>LEN(X$) or if X$ is null or if Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions or string literals.

Example:

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

NOTE: If I=0 is specified, error message "ILLEGAL ARGUMENT IN <line number>" will be returned.

3.19 INT

Format: INT(X)

Versions: 8K, Extended, Disk

Action: Returns the largest integer <=X.

Examples:

```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

See the FIX and CINT functions which also return integer values.

3.20 LEFT$

Format: LEFT$(X$,I)

Versions: 8K, Extended, Disk

Action: Returns a string comprised of the leftmost I characters of X$. I must be in the range 0 to 255. If I is greater than LEN(X$), the entire string (X$) will be returned. If I=0, the null string (length zero) is returned.

Example:

```
10 A$ = "BASIC-80"
20 B$ = LEFT$(A$,5)
30 PRINT B$
BASIC
Ok
```

Also see the MID$ and RIGHT$ functions.

3.21 LEN

Format: LEN(X$)

Versions: 8K, Extended, Disk

Action: Returns the number of characters in X$. Non-printing characters and blanks are counted.

Example:

```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
 16
Ok
```

3.22 LOC

Format: LOC(<file number>)

Version: Disk

Action: With random disk files, LOC returns the record number just read or written from a GET or PUT. If the file was opened but no disk I/O has been performed yet, LOC returns a 0. With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was OPENed.

Example:

```
200 IF LOC(1)>50 THEN STOP
```

3.23 LOG

Format: LOG(X)

Versions: 8K, Extended, Disk

Action: Returns the natural logarithm of X. X must be greater than zero.

Example:
```
PRINT LOG(45/7)
 1.86075
Ok
```

3.24 LPOS

Format: LPOS(X)

Versions: Extended, Disk

Action: Returns the current position of the line printer print head within the line printer buffer. Does not necessarily give the physical position of the print head. X is a dummy argument.

Example:
```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

3.25 MID$

Format: MID$(X$,I[,J])

Versions: 8K, Extended, Disk

Action: Returns a string of length J characters from X$ beginning with the Ith character. I and J must be in the range 1 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X$), MID$ returns a null string.

Example:

```
LIST
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
Ok
RUN
GOOD EVENING
Ok
```

Also see the LEFT$ and RIGHT$ functions.

NOTE: If I=0 is specified, error message "ILLEGAL ARGUMENT IN <line number>" will be returned.

3.26 MKI$, MKS$, MKD$

Format:

```
MKI$(<integer expression>)
MKS$(<single precision expression>)
MKD$(<double precision expression>)
```

Version: Disk

Action: Convert numeric values to string values. Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Example:

```
90 AMT=(K+T)
100 FIELD #1, 8 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
 .
 .
```

See also CVI, CVS, CVD, Section 3.9 and Appendix B.

3.27 OCT$

Format: OCT$(X)

Versions: Extended, Disk

Action: Returns a string which represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Example:

```
PRINT OCT$(24)
 30
Ok
```

See the HEX$ function for hexadecimal conversion.

3.28 PEEK

Format: PEEK(I)

Versions: 8K, Extended, Disk

Action: Returns the byte (decimal integer in the range 0 to 255) read from memory location I. With the 8K version of BASIC-80, I must be less than 32768. To PEEK at a memory location above 32768, subtract 65536 from the desired address. With Extended and Disk BASIC-80, I must be in the range 0 to 65536. PEEK is the complementary function to the POKE statement, Section 2.48.

Example:

```
A=PEEK(&H5A00)
```

3.29 POS

Format: POS(I)

Versions: 8K, Extended, Disk

Action: Returns the current cursor position. The leftmost position is 1. X is a dummy argument.

Example:
```
IF POS(X)>60 THEN PRINT CHR$(13)
```

Also see the LPOS function.

3.30 RIGHT$

Format: RIGHT$(X$,I)

Versions: 8K, Extended, Disk

Action: Returns the rightmost I characters of string X$. If I=LEN(X$), returns X$. If I=0, the null string (length zero) is returned.

Example:
```
10 A$="DISK BASIC-80"
20 PRINT RIGHT$(A$,8)
RUN
BASIC-80
Ok
```

Also see the MID$ and LEFT$ functions.

3.31 RND

Format: RND[(X)]

Versions: 8K, Extended, Disk

Action: Returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see RANDOMIZE, Section 2.53). However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example:

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 24  30  31  51  5
Ok
```

3.32 SGN

Format: SGN(X)

Versions: 8K, Extended, Disk

Action: If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

Example: ON SGN(X)+2 GOTO 100,200,300 branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive.

3.33 SIN

Format: SIN(X)

Versions: 8K, Extended, Disk

Action: Returns the sine of X in radians. SIN(X) is calculated in single precision. COS(X)=SIN(X+3.14159/2).

Example:
```
PRINT SIN(1.5)
 .997495
Ok
```

3.34 SPACE$

Format: SPACE$(X)

Versions: Extended, Disk

Action: Returns a string of spaces of length X. The expression X is rounded to an integer and must be in the range 0 to 255.

Example:
```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
   2
    3
     4
      5
Ok
```

Also see the SPC function.

3.35 SPC

Format: SPC(I)

Versions: 8K, Extended, Disk

Action: Prints I blanks on the terminal. SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255. A';' is assumed to follow the SPC(I) command.

Example:
```
PRINT "OVER" SPC(15) "THERE"
OVER               THERE
Ok
```

Also see the SPACE$ function.

3.36 SQR

Format: SQR(X)

Versions: 8K, Extended, Disk

Action: Returns the square root of X. X must be >=0.

Example:
```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.16228
 15            3.87298
 20            4.47214
 25            5
Ok
```

3.37 STR$

Format: STR$(X)

Versions: 8K, Extended, Disk

Action: Returns a string representation of the value of X.

Example:

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
      .
      .
      .
```

Also see the VAL function.

3.38 STRING$

Formats: STRING$(I,J)
STRING$(I,X$)

Versions: Extended, Disk

Action: Returns a string of length I whose characters all have ASCII code J or the first character of X$.

Example:

```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

3.39 TAB

Format: TAB(I)

Versions: 8K, Extended, Disk

Action: Spaces to position I on the terminal. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

Example:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
RUN
NAME                    AMOUNT

G. T. JONES             $25.00
Ok
```

3.40 TAN

Format: TAN(X)

Versions: 8K, Extended, Disk

Action: Returns the tangent of X in radians. TAN(X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:

```
10 Y = Q*TAN(X)/2
```

3.41 USR

Format : USR[<digit>](X)

Versions: 8K, Extended, Disk

Action: Calls the user's assembly language subroutine with the argument X. <digit> is allowed in the Extended and Disk versions only. <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. See Appendix x.

Example:

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
    .
    .
    .
```

3.42 VAL

Format: VAL(X$)

Versions: 8K, Extended, Disk

Action: Returns the numerical value of string X$. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For example,

VAL(" -3)

returns -3.

Example:

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN
PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815 THEN
PRINT NAME$ TAB(25) "LONG BEACH"
    .
    .
    .
```

See the STR$ function for numeric to string conversion.

3.43 VARPTR

Format 1: VARPTR(<variable name>)

Versions: Extended, Disk

Format 2: VARPTR(#<file number>)

Version: Disk

Action: Format 1: Returns the address of the first byte of data identified with <variable name>. A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Any type variable name may be used (numeric, string, array), and the address returned will be an integer in the range 32767 to -32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

NOTE: All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Format 2: For sequential files, returns the starting address of the disk I/O buffer assigned to <file number>. For random files, returns the address of the FIELD buffer assigned to <file number>.

In Standalone Disk BASIC, VARPTR(#<file number>) returns the first byte of the file block. See Appendix H.

Example:

```
100 X=USR(VARPTR(Y))
```

APPENDIX A

New Features in BASIC-80, Release 5.0

The execution of BASIC programs written under Microsoft BASIC, release 4.51 and earlier may be affected by some of the new features in release 5.0. Before attempting to run such programs, check for the following:

1. New reserved words: CALL, CHAIN, COMMON, WHILE, WEND, WRITE, OPTION BASE, RANDOMIZE.

2. Conversion from floating point to integer values results in rounding, as opposed to truncation. This affects not only assignment statements (e.g., I%=2.5 results in I%=3), but also affects function and statement evaluations (e.g., TAB(4.5) goes to the 5th position, A(1.5) yeilds A(2), and X=11.5 MOD 4 yields 0 for X).

3. The body of a FOR...NEXT loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step. See Section 2.22.

4. Division by zero and overflow no longer produce fatal errors. See Section 1.8.1.2.

5. The RND function has been changed so that RND with no argument is the same as RND with a positive argument. The RND function generates the same sequence of random numbers with each RUN, unless RANDOMIZE is used. See Sections 2.53 and 3.30.

6. The rules for PRINTing single precision and double precision numbers have been changed. See Section 2.49.

7. String space is allocated dynamically, and the first argument in a two-argument CLEAR statement sets the end of memory. The second argument sets the amount of stack space. See Section 2.4.

8. Responding to INPUT with too many or too few items, or with non-numeric characters instead of digits, causes the message "?Redo from start" to be printed. If a single variable is requested, a carriage return may be entered to indicate the default values of 0 for numeric input or null for string input. However, if more than one variable is requested, entering a carriage return will cause the "?Redo from start" message to be printed because too few items were entered. No assignment of input values is made until an acceptable response is given.

9. There are two new field formatting characters for use with PRINT USING. An ampersand is used for variable length string fields, and an underscore signifies a literal character in a format string.

10. If the expression supplied with the WIDTH statement is 255, BASIC uses an "infinite" line width, that is, it does not insert carriage returns. WIDTH LPRINT may be used to set the line width at the line printer. See Section 2.66.

11. The at-sign and underscore are no longer used as editing characters.

12. Variable names are significant up to 40 characters and can contain embedded reserved words. However, reserved words must now be delimited by spaces. To maintain compatibility with earlier versions of BASIC, spaces will be automatically inserted between adjoining reserved words and variable names. WARNING: This insertion of spaces may cause the end of a line to be truncated if the line length is close to 255 characters.

13. BASIC programs may be saved in a protected binary format. See SAVE, Section 2.60.

APPENDIX B

BASIC-80 Disk I/O

Disk I/O procedures for the beginning BASIC-80 user are examined in this appendix. If you are new to BASIC-80 or if you're getting disk related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements for filenames. The CP/M operating system will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE or LOAD command.

B.1 PROGRAM FILE COMMANDS

Here is a review of the commands and statements used in program file manipulation.

SAVE <filename>[,A]	Writes to disk the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)
LOAD <filename>[,R]	Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files.

RUN <filename>[,R]	RUN <filename> loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.
MERGE <filename>	Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.
KILL <filename>	Deletes the file from the disk. <filename> may be a program file, or a sequential or random access data file.
NAME <old filename> AS<new filename>	To change the name of a disk file, execute the NAME statement, NAME <oldfile> AS <newfile>. NAME may be used with program files, random files, or sequential files.

B.2 PROTECTED FILES

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

B.3 DISK DATA FILES - SEQUENTIAL AND RANDOM I/O

There are two types of disk data files that may be created and accessed by a BASIC-80 program: sequential files and random access files.

B.3.1 Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with sequential files are:

```
OPEN    PRINT#          INPUT#         WRITE#
        PRINT# USING    LINE INPUT#

CLOSE   EOF    LOC
```

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode. `OPEN "O",#1,"DATA"`

2. Write data to the file using the PRINT# statement. (WRITE# may be used instead.) `PRINT#1,A$;B$;C$`

3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode. `CLOSE #1` `OPEN "I",#1,"DATA"`

4. Use the INPUT# statement to read data from the sequential file into the program. `INPUT#1,X$,Y$,Z$`

Program B-1 is a short program that creates a sequential file, "DATA", from information you input at the terminal.

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

PROGRAM B-1 - CREATE A SEQUENTIAL DATA FILE

Now look at Program B-2. It accesses the file "DATA" that was created in Program B-1 and displays the name of everyone hired in 1978.

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

PROGRAM B-2 - ACCESSING A SEQUENTIAL FILE

Program B-2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed. A sector is a 128-byte block of data.

B.3.1.1 Adding Data To A Sequential File -

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. The following procedure can be used to add data to an existing file called "NAMES".

1. OPEN "NAMES" in "I" mode.

2. OPEN a second file called "COPY" in "O" mode.

3. Read in the data in "NAMES" and write it to "COPY".

4. CLOSE "NAMES" and KILL it.

5. Write the new information to "COPY".

6. Rename "COPY" as "NAMES" and CLOSE.

7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program B-3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# will read in characters from the disk until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

PROGRAM B-3 - ADDING DATA TO A SEQUENTIAL FILE

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

B.3.2 Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk -- it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

OPEN FIELD LSET/RSET GET

PUT CLOSE LOC

MKI$ CVI
MKS$ CVS
MKD$ CVD

B.3.2.1 Creating A Random File -
The following program steps are required to create a random file.

1.	OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.	OPEN "R",#1,"FILE",32
2.	Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.	FIELD #1 20 AS N$, 4 AS A$, 8 AS P$
3.	Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.	LSET N$=X$ LSET A$=MKS$(AMT) LSET P$=TEL$
4.	Write the data from the buffer to the disk using the PUT statement.	PUT #1,CODE%

Look at Program B-4. It takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

NOTE

Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

PROGRAM B-4 - CREATE A RANDOM FILE

B.3.2.2 Access A Random File -
The following program steps are required to access a random file:

1. OPEN the file in "R" mode. `OPEN "R",#1,"FILE",32`

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file. `FIELD #1 20 AS N$, 4 AS A$, 8 AS P$`

NOTE:
In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer. — `GET #1,CODE%`

4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values. — `PRINT N$` / `PRINT CVS(A$)`

Program B-5 accesses the random file "FILE" that was created in Program B-4. By inputting the three-digit code at the terminal, the information associated with that code is read from the file and displayed.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

PROGRAM B-5 - ACCESS A RANDOM FILE

The LOC function, with random files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file#1 is higher than 50.

Program B-6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
        "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:
        IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;" IN STOCK":GOTO 600
630 Q%=Q%-S%
```

```
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
         " REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";
          CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER":
         GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

PROGRAM B-6 - INVENTORY

APPENDIX C

Assembly Language Subroutines

All versions of BASIC-80 have provisions for interfacing with assembly language subroutines. The USR function allows assembly language subroutines to be called in the same way BASIC's intrinsic functions are called.

NOTE

> The addresses of the DEINT, GIVABF, MAKINT and FRCINT routines are stored in locations that must be supplied individually for different implementations of BASIC.

C.1 MEMORY ALLOCATION

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s). BASIC uses all memory available from its starting location up, so only the topmost locations in memory can be set aside for user subroutines.

When an assembly language subroutine is called, the stack pointer is set up for 8 levels (16 bytes) of stack storage. If more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

The assembly language subroutine may be loaded into memory by means of the system monitor, or the BASIC POKE statement, or (if the user has the MACRO-80 or FORTRAN-80 package) routines may be assembled with MACRO-80 and loaded using LINK-80.

C.2 USR FUNCTION CALLS - 8K BASIC

The starting address of the assembly language subroutine must be stored in USRLOC, a two-byte location in memory that is supplied individually with different implementations of BASIC-80. With 8K BASIC, the starting address may be POKEd into USRLOC. Store the low order byte first, followed by the high order byte.

The function USR will call the routine whose address is in USRLOC. Initially USRLOC contains the address of ILLFUN, the routine that gives the "Illegal function call" error. Therefore, if USR is called without changing the address in USRLOC, an "Illegal function call" error results.

The format of a USR function call is

```
USR(argument)
```

where the argument is a numeric expression. To obtain the argument, the assembly language subroutine must call the routine DEINT. DEINT places the argument into the D,E register pair as a 2-byte, 2's complement integer. (If the argument is not in the range -32768 to 32767, an "Illegal function call" error occurs.)

To pass the result back from an assembly language subroutine, load the value in register pair [A,B], and call the routine GIVABF. If GIVABF is not called, USR(X) returns X. To return to BASIC, the assembly language subroutine must execute a RET instruction.

For example, here is an assembly language subroutine that multiplies the argument by 2:

```
USRSUB: CALL DEINT      ;put arg in D,E
        XCHG            ;move arg to H,L
        DAD H           ;H,L=H,L+H,L
        MOV A,H         ;move result to A,B
        MOV B,L
        JMP GIVABF      ;pass result back and RETurn
```

Note that valid results will be obtained from this routine for arguments in the range -16384<=x<=16383. The single instruction JMP GIVABF has the same effect as:

```
CALL GIVABF
RET
```

To return additional values to the program, load them into memory and read them with the PEEK function.

There are several methods by which a program may call more than one USR routine. For example, the starting address of each routine may be POKEd into USRLOC prior to each USR call, or the argument to USR could be an index into a table of USR routines.

C.3 USR FUNCTION CALLS - EXTENDED AND DISK BASIC

In the Extended and Disk versions, the format of the USR function is

USR[<digit>](argument)

where DIGIT> is from 0 to 9 and the argument is any numeric or string expression. <digit> specifies which USR routine is being called, and corresponds with the digit supplied in the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. The address given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the type of argument that was given. The value in A may be one of the following:

Value in A	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Single precision floating point number
8	Double precision floating point number

If the argument is a number, the [H,L] register pair points to the Floating Point Accumulator (FAC) where the argument is stored.

If the argument is an integer:

FAC-3 contains the lower 8 bits of the argument and
FAC-2 contains the upper 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-3 contains the lowest 8 bits of mantissa and

FAC-2 contains the middle 8 bits of mantissa and FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative). FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

If the argument is a double precision floating point number:

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the [D,E] register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

CAUTION: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. Example:

```
A$ = "BASIC-80"+""
```

This will copy the string literal into string space and will prevent alteration of program text during a subroutine call.

Usually, the value returned by a USR function is the same type (integer, string, single precision or double precision) as the argument that was passed to it. However, calling the MAKINT routine returns the integer in [H,L] as the value of the function, forcing the value returned by the function to be integer. To execute MAKINT, use the following sequence to return from the subroutine:

```
PUSH    H       ;save value to be returned
LHLD    xxx     ;get address of MAKINT routine
XTHL            ;save return on stack and
                ;get back [H,L]
RET             ;return
```

Also, the argument of the function, regardless of its type, may be forced to an integer by calling the FRCINT routine to get the integer value of the argument in [H,L]. Execute the following routine:

```
   LXI     H       ;get address of subroutine
                   ;continuation
   PUSH    H       ;place on stack
   LHLD    xxx     ;get address of FRCINT
   PCHL
SUB1: . . . . .
```

C.4 CALL STATEMENT

Extended and Disk BASIC-80 user function calls may also be made with the CALL statement. The calling sequence used is the same as that in Microsoft's FORTRAN and COBOL compilers.

A CALL statement with no arguments generates a simple "CALL" instruction. The corresponding subroutine should return via a simple "RET." (CALL and RET are 8080 opcodes - see an 8080 reference manual for details.)

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing the parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).

2. If the number of parameters is greater than 3, they are passed as follows:

   1. Parameter 1 in HL.

   2. Parameter 2 in DE.

   3. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that, with this scheme, the subroutine must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for the correct number or type of parameters.

If the subroutine expects more than 3 parameters, and needs to transfer them to a local data area, there is a system subroutine which will perform this transfer. This argument transfer routine is named $AT (located in the FORTRAN library, FORLIB.REL), and is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2). The subroutine is

responsible for saving the first two parameters before calling $AT. For example, if a subroutine expects 5 parameters, it should look like:

```
SUBR:   SHLD    P1      ;SAVE PARAMETER 1
        XCHG
        SHLD    P2      ;SAVE PARAMETER 2
        MVI     A,3     ;NO. OF PARAMETERS LEFT
        LXI     H,P3    ;POINTER TO LOCAL AREA
        CALL    $AT     ;TRANSFER THE OTHER 3 PARAMETERS
        .
        .
        .
        [Body of subroutine]
        .
        .
        .
        RET             ;RETURN TO CALLER
P1:     DS      2       ;SPACE FOR PARAMETER 1
P2:     DS      2       ;SPACE FOR PARAMETER 2
P3:     DS      6       ;SPACE FOR PARAMETERS 3-5
```

A listing of the argument transfer routine $AT follows.

```
;        ARGUMENT TRANSFER
;[B,C]   POINTS TO 3RD PARAM.
;[H,L]   POINTS TO LOCAL STORAGE FOR PARAM 3
;[A]     CONTAINS THE # OF PARAMS TO XFER (TOTAL-2)


         ENTRY   $AT
$AT:     XCHG              ;SAVE [H,L] IN [D,E]
         MOV     H,B
         MOV     L,C       ;[H,L] = PTR TO PARAMS
AT1:     MOV     C,M
         INX     H
         MOV     B,M
         INX     H         ;[B,C] = PARAM ADR
         XCHG              ;[H,L] POINTS TO LOCAL STORAGE
         MOV     M,C
         INX     H
         MOV     M,B
         INX     H         ;STORE PARAM IN LOCAL AREA
         XCHG              ;SINCE GOING BACK TO AT1
         DCR     A         ;TRANSFERRED ALL PARAMS?
         JNZ     AT1       ;NO, COPY MORE
         RET               ;YES, RETURN
```

When accessing parameters in a subroutine, don't forget that they are pointers to the actual arguments passed.

NOTE

> It is entirely up to the programmer to see to it that the arguments in the calling program match in number, type, and length with the parameters expected by the subroutine. This applies to BASIC subroutines, as well as those written in assembly language.

C.5 INTERRUPTS

Assembly language subroutines can be written to handle interrupts. All interrupt handling routines should save the stack, register A-L and the PSW. Interrupts should always be re-enabled before returning from the subroutine., since an interrupt automatically disables all further interrupts once it is received. The user should be aware of which interrupt vectors are free in the particular version of BASIC that has been supplied. (Note to CP/M users: In CP/M BASIC, all interrupt vectors are free.)

[illegible]

[illegible]

[illegible]

[illegible]

APPENDIX D

BASIC-80 with the CP/M Operating System

The CP/M version of BASIC-80 (MBASIC) is supplied on a standard size 3740 single density diskette. The name of the file is MBASIC.COM. (A 28K or larger CP/M system is recommended.)

To run MBASIC, bring up CP/M and type the following:

```
A>MBASIC <carriage return>
```

The system will reply:

```
xxxx Bytes Free
BASIC-80 Version 5.0
(CP/M Version)
Copyright 1978 (C) by Microsoft
Created: dd-mmm-yy
Ok
```

MBASIC is the same as Disk BASIC-80 as described in this manual, with the following exceptions:

D.1 INITIALIZATION

The initialization dialog has been replaced by a set of options which are placed after the MBASIC command to CP/M. The format of the command line is:

```
A>MBASIC [<filename>][/F:<number of files>][/M:<highest memory location>]
      [/S:<maximum record size>]
```

If <filename> is present, MBASIC proceeds as if a RUN <filename> command were typed after initialization is complete. A default extension of .BAS is used if none is supplied and the filename is less than 9 characters long. This allows BASIC programs to be executed in batch mode using the SUBMIT facility of CP/M. Such programs should include a SYSTEM statement (see below) to return to CP/M when they have finished, allowing the next program in the

batch stream to execute.

If /F:<number of files> is present, it sets the number of disk data files that may be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes of memory. If the /F option is omitted, the number of files defaults to 3.

The /M:<highest memory location> option sets the highest memory location that will be used by MBASIC. In some cases it is desirable to set the amount of memory well below the CP/M's FDOS to reserve space for assembly language subroutines. In all cases, <highest memory location> should be below the start of FDOS (whose address is contained in locations 6 and 7). If the /M option is omitted, all memory up to the start of FDOS is used.

/S:<maximum record size> may be added at the end of the command line to set the maximum record size for use with random files. The default record size is 128 bytes.

NOTE

<number of files>, <highest memory location>, and <maximum record size> are numbers that may be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

Examples:

A>MBASIC PAYROLL.BAS	Use all memory and 3 files, load and execute PAYROLL.BAS.
A>MBASIC INVENT/F:6	Use all memory and 6 files, load and execute INVENT.BAS.
A>MBASIC /M:32768	Use first 32K of memory and 3 files.
A>MBASIC DATACK/F:2/M:&H9000	Use first 36K of memory, 2 files, and execute DATACK.BAS.

D.2 DISK FILES

Disk filenames follow the normal CP/M naming conventions. All filenames may include A: or B: as the first two characters to specify a disk drive, otherwise the currently selected drive is assumed. A default extension of .BAS is

used on LOAD, SAVE, MERGE and RUN <filename> commands if no "." appears in the filename and the filename is less than 9 characters long.

For systems with CP/M 2.x, large random files are supported. The maximum logical record number is 32767. If a record size of 256 is specified, then files up to 8 megabytes can be accessed.

D.3 FILES COMMAND

Format: FILES[<filename>]

Purpose: To print the names of files residing on the current disk.

Remarks: If <filename> is omitted, all the files on the currently selected drive will be listed. <filename> is a string formula which may contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the filename or extension will match any file or any extension.

Examples:

```
FILES
FILES "*.BAS"
FILES "B:*.*"
FILES "TEST?.BAS"
```

D.4 RESET COMMAND

Format: RESET

Purpose: To close all disk files and write the directory information to a diskette before it is removed from a disk drive.

Remarks: Always execute a RESET command before removing a diskette from a disk drive. Otherwise, when the diskette is used again, it will not have the current directory information written on the directory track.

RESET closes all open files on all drives and writes the directory track to every diskette with open files.

D.5 LOF FUNCTION

Format: LOF(<file number>)

Action: Returns the number of records present in the last extent read or written. If the file does not exceed one extent (128 records), then LOF returns the true length of the file.

Example: 110 IF NUM%>LOF(1) THEN PRINT "INVALID ENTRY"

D.6 EOF

With CP/M, the EOF function may be used with random files. If a GET is done past the end of file, EOF will return -1. This may be used to find the size of a file using a binary search or other algorithm.

D.7 MISCELLANEOUS

1. CSAVE and CLOAD are not implemented.

2. To return to CP/M, use the SYSTEM command or statement. SYSTEM closes all files and then performs a CP/M warm start. Control-C always returns to MBASIC, not to CP/M.

3. FRCINT is at 103 hex and MAKINT is at 105 hex. (Add 1000 hex for ADDS versions, 4000 for SBC CP/M versions.)

APPENDIX E

BASIC-80 with the ISIS-II Operating System

With ISIS-II, BASIC-80 is the same as described in this manual, with the following exceptions:

E.1 INITIALIZATION

The initialization dialog has been replaced by a set of options which are placed after the MBASIC command to ISIS-II. The format of the command line is:

```
-MBASIC [<filename>][/F:<number of files>][/M:<highest memory location>]
     [/S:<maximum record size>]
```

If <filename> is present, BASIC proceeds as if a RUN <filename> command were typed after initialization is complete. A default extension of .BAS is used if none is supplied.

If /F:<number of files> is present, it sets the number of disk data files that may be open at any one time during the execution of a BASIC program. The maximum is six and the default is three. The /M:<highest memory location> option sets the highest memory location that will be used by BASIC. Use this option to reserve memory locations above BASIC for assembly language subroutines. /S:<maximum record size> may be added at the end of the command line to set the maximum record size for use with random files. The default record size is 128 bytes.

At initialization, the system will reply:

```
xxxx Bytes Free
BASIC-80 Version x.x
(ISIS-II Version)
Copyright 1978 (C) by Microsoft
```

E.2 LINE PRINTER I/O

To send output to the printer during execution of a BASIC program, open the line printer as if it were a disk file:

```
50 N=4
100 OPEN "O",N,":LP:"
 .
 .
 .
120 PRINT #N,A,B,C
```

Since BASIC buffers disk I/O, you may want to force buffers out by CLOSEing the printer channel.

To LIST a program on the line printer, use:

```
SAVE":LP:",A
```

E.3 ATTRIB STATEMENT

In ISIS-II BASIC-80, the ATTRIB statement sets file attributes. The format of the statement is:

```
ATTRIB <filename string>,<attribute string>
```

The attribute string consists of F, W, S or I for the attribute, followed by a 1 to set the attribute or a 0 to reset.

Examples:

```
ATTRIB "INFO.DAT","W1"
ATTRIB "GHOST.BAS","I1"
ATTRIB ":F1:SYSFIL","W1F1S1I1"
ATTRIB A$,B$
```

E.4 MISCELLANEOUS

Note these other differences for ISIS-II BASIC:

1. MAKINT is located at 3903 hex, and GIVINT is located at 3905 hex.

2. There is no FILES command in ISIS-II BASIC. Filenames do <u>not</u> default to .BAS on SAVEs, LOADs, and MERGEs.

APPENDIX F

BASIC-80 with the TEKDOS Operating System

The operation of BASIC-80 with the TEKDOS operating system is the same as described in this manual with the following exceptions:

1. At initialization, BASIC asks MEMORY SIZE? If you respond with a carriage return, BASIC will use all available memory. If you respond with a memory location (in decimal), BASIC will use memory only up to that location. This lets you reserve space at the top of memory for assembly language subroutines.

2. The number of disk files that may be open at one time defaults to 5.

3. LPRINT and LLIST are not implemented. Instead, open a file to the printer.

4. TEKDOS does not support random disk I/O. The corresponding BASIC-80 statements (PUT, GET, OPEN"R", etc.) are inoperable under TEKDOS.

5. Control-C works only once due to a bug in TEKDOS. If you interrupt a running program or a LIST command with Control-C, BASIC appears to be in "single statement" mode. To clear this condition, exit BASIC with a SYSTEM command and re-enter BASIC with an XEQ BASIC. Avoid using the AUTO command, since it requires a Control-C to return to BASIC command level.

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

APPENDIX G

BASIC-80 with the INTEL SBC and MDS Systems

G.1 INITIALIZATION

The paper tape of BASIC-80 supplied for SBC and MDS systems is in Intel-compatible hex format. Use the monitor's R command to load the tape, then execute the G command to start BASIC-80. The command is:

```
.G4000
```

BASIC will respond:

```
Memory size?
```

If you want BASIC to use all available RAM, just type a carriage return. If you want to reserve space at the top of memory for machine language subroutines, enter the highest memory address (in decimal) that BASIC may use.

```
Terminal Width?
```

(8K versions only) Respond with the number of characters for the output line width in PRINT statements. The default is 72 characters. (Extended versions use WIDTH command.)

```
Want SIN-COS-TAN-ATN?
```

Type Y to retain these functions, type N to delete them, or type A to delete ATN only.

G.2 SUBROUTINE ADDRESSES

In the 8K version of SBC and MDS BASIC-80, DEINT is located at 0043 hex and GIVABF is located at 0045 hex. USRLOC is at xxxx hex. In the Extended version, FRCINT is located at xxxx hex, and MAKINT is located at xxxx hex.

G.3 LLIST AND LLPRINT

LLIST and LPRINT are not implemented.

APPENDIX H

Standalone Disk BASIC

Standalone Disk BASIC is an easily implemented, self-contained version of BASIC-80 that runs on almost any 8080 or Z80 based disk hardware without an operating system. Standalone Disk BASIC incorporates several unique disk I/O methods that make faster and more efficient use of disk access and storage.

Random access with Standalone BASIC is faster than other disk operating systems because the file allocation table is kept in memory and updated periodically on the diskette. Therefore, there is no need for index blocks for random files, and there is no need to distinguish between random and sequential files. Because there are no index blocks, there is no large per-file-overhead either in memory or on disk. Binary SAVEs and LOADs are also faster because they are optimized by cluster, i.e., an entire cluster is read or written at one time, instead of a single sector.

To initialize Standalone Disk BASIC, insert the BASIC diskette and power up the system. In one- or two-drive systems, BASIC asks if there are two drives. In systems with more than two drives, BASIC asks for the number of drives. BASIC then asks how many files, i.e., how many disk files may be open at one time. Answer with a number from 0 to 15, or, for a default of 1 file per drive, just enter a carriage return.

The operation of Standalone Disk BASIC is the same as Disk BASIC-80 as described in this manual, with the following exceptions:

H.1 FILENAMES

The format for disk filenames is:

```
[drive#:]filename[.extension]
```

The first drive is 1.

Disk filenames are six characters with an optional three-character extension that is preceded by a decimal point. If a decimal point appears in a filename after fewer than six characters, the name is blank-filled to six characters and the next three characters are the extension. If the filename is six or fewer characters with no decimal point, there is no extension. If the filename is more than six characters, BASIC inserts a decimal point after the sixth character and uses the next three characters as an extension. (Any additional characters are ignored.)

H.2 DISK FILES

The FILES command prints the names of the files residing on a disk. The format is: [L]FILES[<drive number>]
LFILES outputs to the line printer. In addition to the filename, the size of each file, in clusters, is output. A cluster is the minimum unit of allocation for a file -- it is one-half of a track. Filenames of files created with OPEN or ASCII SAVE are listed with a space between the name and extension. Filenames of binary files created with binary SAVE are listed with a decimal point between the name and extension. The protected file option with SAVE is not supported in Standalone Disk BASIC.

H.3 FPOS

The FPOS function:

```
FPOS(<file number>)
```

is the same as BASIC-80's LOC function except it returns the number of the physical sector where <filenumber> is located. (BASIC-80's LOC function and CP/M BASIC-80's LOF function are also implemented.)

H.4 DSKI$/DSKO$

The DSKO$ statement:

```
DSKO$<drive>,<track>,<sector>,<string expression>
```

writes the string on the specified sector. The maximum length for the string is 128 characters. A string of fewer than 128 characters is zero-filled at the end to 128 characters.

DSKI$ is the complementary function to the DSKO$ statement. DSKI$ returns the contents of a sector to a string variable name. The format is:

```
DSKI$(<drive>,<track>,<sector>)
```

Example: A$=DSKI$(0,I,J)

H.5 MOUNT COMMAND

Before a diskette can be used for file operations (i.e., any disk I/O besides DSKI$, DSKO$, or IBM or USR modes), it must be MOUNTed. The format of the command is:

```
MOUNT[<drive>[,<drive>...]]
```

MOUNT with no arguments mounts all drives. When a diskette is mounted, BASIC reads the File Allocation Table (see Section H.11.2) from the diskette into memory and checks it for errors. If there are no errors, the disk is mounted. If an error is found, BASIC reads one or both of the back-up allocation tables from the diskette in an attempt to mount the disk; and a warning message, "x copies of allocation bad on drive y", is issued. x is 1 or 2 and y is the drive number. When a warning occurs, it is a good idea to make a new copy of the diskette. If all copies of the allocation table are bad or if a free entry is encountered in the file chain, a fatal error--"bad allocation table"--is given and the diskette will not be mounted.

While a disk is mounted, BASIC occasionally writes the allocation table to the directory track, but it does not check for errors unless the read after write attribute is set for that drive (see SET statement).

H.6 REMOVE COMMAND

REMOVE is the complement of MOUNT. Before a diskette can be taken out of the drive, a REMOVE command must be executed. The format of the command is:

```
REMOVE[<drive>[,<drive>...]]
```

REMOVE writes three copies of the current allocation table to disk and follows the same error-check procedure as MOUNT. MOUNT and REMOVE replace the RESET command that is in BASIC-80.

NOTE

ALWAYS do a REMOVE before taking a diskette out of a drive. If you do not, the diskette you took out will not have an updated and checked allocation table, and the data on the next diskette inserted will be destroyed when the wrong allocation table is written to the directory track.

H.7 SET STATEMENT

The SET statement determines the attributes of the currently mounted disk drive, a currently open file, or a file that need not be open. The format of the SET statement is:

SET<drive> | #<file> | <filename>,<attribute string>

<attribute string> is a string of characters that determines what attributes are set. Any characters other than the following are ignored:

R	Read after write
P	Write protect
E	EBCDIC conversion (if available)

Attributes are assigned in the following order:

1. MOUNT command
   When a MOUNT is done for a particular drive, the first byte of the information sector on the diskette (track 35, sector 20 for floppy; track 18, sector 13 for minifloppy) contains the attributes for the disk. (octal values: R=100, P=20, E=40)

2. SET<drive>,<attribute string> Statement
   This statement sets the current attributes for the disk, in memory, while it is mounted. The attributes are not permanently recorded and apply only while the disk is mounted.

3. When a file is created, the permanent file attributes recorded on the disk will be the same as the current drive attributes.

4. SET<filename>,<attribute string> Statement
   This statement changes the permanent file attributes that are stored in the directory entry for that file. It does not affect the drive attributes.

5. When an existing file is OPENed, the attributes of the file number are those of the directory entry.

6. SET#<file number>,<attribute string> Statement
   This statement changes the attributes for that file number but does not change the directory entry.

Examples:

SET 1,"R"	Force read after write checking on all output to drive 1
SET #1,"R"	Force read after write for all output to file 1 while it is open
SET #1,"P"	Give write protect error if any output is attempted to file 1
SET "TEST","P"	Protect TEST from deletion and modification
SET 1,""	Turn off all attributes for drive 1

H.8 ATTR$ FUNCTION

ATTR$ returns a string of the current attributes for a drive, currently open file, or file that need not be open. The format of ATTR$ is:

    ATTR$(<drive> | #<file number> | <filename>)

For example:

```
SET 1,"R":A$=ATTR$(1):PRINT A$
R
Ok
```

H.9 OPEN STATEMENT

The format for the OPEN statement in Standalone BASIC is:

    OPEN <filename> [FOR <mode>] AS [#]<file number>

where <mode> is one of the following:

    INPUT
    OUTPUT
    APPEND
    IBM
    USR

The mode determines only the initial positioning within the file and the actions to be taken if the file does not exist. The action taken in each mode is:

INPUT The initial position is at the start of the file. An error is returned if the file is not found.

OUTPUT The initial position is at the start of the file. A new file is always created.

APPEND The initial position is at the end of the file. An error is returned if the file is not found.

IBM The initial position is after the last DSKI$ or DSKO$. The file is then set up to write contiguous. No file search is done. (The same effect may be achieved in many cases by altering the FORMAT program. See Section H.11.2.1.)

USR Same as IBM mode except, instead of write contiguous, USR0 is called and returns the next track/sector number. The USR0 routine should read the current track/sector from B,C and return the next location in B,C. When USR0 is first called, B,C contains the track and sector number of the previous DSKI$ or DSKO$.

If the FOR <mode> clause is omitted, the initial position is at the start of the file. If the file is not found, it is created.

Note that variable length records are not supported in Standalone Disk BASIC. All records are 128 bytes in length.

USR mode is especially useful for creating diskettes that require sector mapping. This is the case if the diskette is intended for use on another system, for example, a CP/M system. Instead of opening the file for write contiguous (IBM mode), the USR0 routine may be used to map the sectors logically, as required by the other system.

When a file is OPENed FOR APPEND, the file mode is set to APPEND and the record number is set to the last record of the file. The program may subsequently execute disk I/O statements that move the pointer elsewhere in the file. When the last record is read, the file mode is reset to FILE and the pointer is left at the end of the file. Then, if you wish to append another record, execute:

```
GET#n,LOF(n)
```

This positions the pointer at the end of the file in preparation for appending.

At any one time, it is possible to have a particular

filename OPEN under more than one file number. This allows different attributes to be used for different purposes. Or, for program clarity, you may wish to use different file numbers for different methods of access. Each file number has a different buffer, so changes made under one file are not accessible to (or affected by) the other numbers until that record is written (e.g., GET#n,LOC(n)).

H.10 DISK I/O

A GET or PUT (i.e., random access) cannot be done on a file that is OPEN FOR IBM or OPEN FOR USR. Otherwise, GET/PUT may be executed along with PRINT#/INPUT# on the same file, which makes midfile updating possible. The statement formats for GET, PUT, PRINT#, and INPUT# are the same as those in BASIC-80. The action of each statement in Standalone BASIC is as follows:

GET — If the "buffer changed" flag is set, write the buffer to disk. Then execute the GET (read the record into the buffer), and reset the position for sequential I/O to the beginning of the buffer.

PUT — Execute the PUT (write the buffer to the specified record number), and set the "sequential I/O is illegal" flag until a GET is done.

INPUT# — If the buffer is empty, write it if the "buffer changed" flag is set, then read the next buffer.

PRINT# — Set the "buffer changed" flag. If the buffer is full, write it to disk. Then, if end of file has not been reached, read the next buffer.

H.10.1 File Format

For a single density floppy, each file requires 137 bytes: 9 bytes plus the 128-byte buffer. Because the File Allocation Table keeps random access information for all files, random and sequential files are identical on the disk. The only distinction is that sequential files have a Control-Z (32 octal) as the last character of the last sector. When this sector is read, it is scanned from the end for a non-zero byte. If this byte is Control-Z, the size of the buffer is set so that a PRINT overwrites this byte. If the byte is not Control-Z, the size is set so the last null seen is overwritten.

Any sequential file can be copied in random mode and remain identical. If a file is written to disk in random mode

(i.e., with PUT instead of PRINT) and then read in sequential mode, it will still have proper end of file detection.

H.11 DISK ALLOCATION INFORMATION

With Standlone Disk BASIC, storage space on the diskette is allocated beginning with the cluster closest to the current position of the head. (This method is optimized for writing. Custom versions can be optimized for reading.) Disk allocation information is placed in memory when the disk is mounted and is periodically written back to the disk. Because this allocation information is kept in memory, there is no need for index blocks for random files, and there is no need to distinguish between random and sequential files.

H.11.1 Directory Format

On the diskette, each sector of the directory track contains eight file entries. Each file entry is 16 bytes long and formatted as follows:

Bytes	Usage
0-8	Filename, 1 to 9 characters. The first character may not be 0 or 255.
9	Attribute: Octal 200 Binary file 100 Force read after write check 40 EBCDIC file 20 Write protected file Excluding 200, these bits are the same for the disk attribute byte which is the first byte of the information sector.
10	Pointer into File Allocation Table to the first cluster of the file's cluster chain.
11-15	Reserved for future expansion.

If the first byte of a filename is zero, that file entry slot is free. If the first byte is 255, that slot is the last occupied slot in the directory, i.e., this flags the end of the directory.

H.11.2 Drive Information

For each disk drive that is MOUNTed, the following information is kept in memory:

1. Attributes
   Drive attributes are read from the information sector when the drive is mounted and may be changed with the SET statement. Current attributes may be examined with the ATTR$ function.

2. Track Number
   This is the current track while the disk is mounted. Otherwise, track number contains 255 as a flag that the disk is not mounted.

3. Modification Counter
   This counter is incremented whenever an entry in the File Allocation Table is changed. After a given number of changes has been made, the File Allocation Table is written to disk.

4. Number of Free Clusters
   This is calculated when the drive is mounted, and updated whenever a file is deleted or a cluster is allocated.

5. File Allocation Table
   The File Allocation Table has a one-byte entry for every cluster allocated on the disk. If the cluster is free, this entry is 255. If the cluster is reserved, this entry is 254. If the cluster is the last cluster of the file, this entry is 300 (octal) plus the number of sectors from this cluster that were used. Otherwise, the entry is a pointer to the next cluster of the file. The File Allocation Table is read into memory when the drive is mounted, and updated:

   1. When a file is deleted

   2. When a file is closed

   3. When modifications to the table total twice the number of sectors in a cluster (this can be changed in custom versions)

   4. When modifications to the table have been made and the disk head is on (or passes) the directory track.

H.11.2.1 FORMAT Program - Before mounting a drive with a new diskette, run BASIC's FORMAT program to initialize the directory (set all bytes to 255), set the information sector to 0, and set all the File Allocation Table entries (except the directory track entry (254)) to "free" (255).

The FORMAT program is:

```
10 CLEAR 1500
20 A$=STRING$(128,255)
30 B$=STRING$(35*2,255)+STRING$(2,254)+STRING$(56,255)
40 FOR S=1 TO 19:DSKO$ 1,35,S,A$:NEXT
50 FOR S=21 TO 25 STEP 2:DSKO$ 1,35,S,B$
60 DSKO$ 1,35,S+1,A$:NEXT
70 DSKO$ 1,35,20,CHR$(0)
```

After running FORMAT and MOUNTing the drive, files will be allocated as usual, i.e., on either side of the directory track.

The FORMAT program may be altered to pre-allocate selected files. For instance, you may wish to use the FORMAT program to pre-allocate files contiguously (as they would be allocated in IBM mode). Then IBM and BASIC files may both exist on the diskette. The altered FORMAT program must also write the name of the file(s) to the directory track (i.e., files 1-8 in sector 1, files 9-16 in sector 2, etc.), so BASIC knows where the files start.

H.11.3 File Block

Each file on the disk has a file block that contains the following information:

1. File Mode (byte 0)
   This is the first byte (byte 0) of the file block, and its location may be read with VARPTR(#filenumber). The location of any other byte in the file block is relative to the file mode byte. The file mode byte is one of the following:

(octal)	
1	Input only
2	Output only
4	File mode
10	Append mode
20	Delete file
40	IBM mode
100	Special format (USR)
200	Binary save

NOTE

> It is not recommended that the user attempt to modify the next four bytes of the File Allocation Table. Many unforeseen complications may result.

2. Pointer to the File Allocation Table entry for the first cluster allocated to the file (+1)

3. Pointer to the File Allocation Table entry for the last cluster accessed (+2)

4. Last sector accessed (+3)

5. Disk number of file (+4)

6. The size of the last buffer read (+5). This is 128 unless the last sector of the file is not full (i.e., Control-Z).

7. The current position in the buffer (+6). This is the offset within the buffer for the next print or input.

8. File flag (+7), is one of the following:

Octal	
100	Read after write check
40	Read/Write EBCDIC, not ASCII (Not available in all versions.)
20	File write protected
10	Buffer changed by PRINT
4	PUT has been done. PRINT/INPUT are errors until a GET is done. (See Section H.10.)
2	Flags buffer is empty

9. Terminal position for TAB function and comma in PRINT statements (+8)

10. Beginning of sector buffer (+9), 128 bytes in length

H.12 ADVANCED USES OF FILE BUFFERS

1. Information may be passed from one program to another by FIELDing it to an unopened file number (not #0). The FIELD buffer is not cleared as long as the file is not OPENed.

2. The FIELDed buffer for an unopened file can also be used to format strings. For example, an 80-character string could be placed into a FIELDed buffer with LSET. The strings could then be accessed as four 20-character strings using their FIELDed variable names. For example:

```
100 FIELD#1, 80 AS A$
200 FIELD#1, 20 AS A1$, 20 AS A2$, 20 AS A3$, 20 AS A4$
300 LINE INPUT "CUSTOMER INFORMATION: ";B$
400 LSET A$=B$
500 PRINT "NAME ";A1$;"SSN: ";A2$
```

3. FIELD#0 may be used as a temporary buffer, but note that this buffer is cleared after each of the following commands: FILES, LOAD, SAVE, MERGE, RUN, DSKO$, MOUNT, OPEN.

4. The effect of PRINT[USING]# into a string may be achieved by printing to a FIELDed buffer and then accessing it without reopening the file. To assure that this temporary buffer is not written to the disk, return the pointer to the beginning of the buffer and reset the "buffer changed" flag as follows:

```
10 OPEN "D" FOR IBM AS 1:REM THIS DOESN'T USE SPACE
20 PRINT USING#1 ...
30 P=PEEK(6+VARPTR(#1)):REM OPTIONAL, TO GET LENGTH OF PRINT
USING
40 FIELD#1 ... AS ...
50 Y=7+VARPTR(#1)
60 POKE Y,PEEK(Y AND &360):REM RESET BUFFER CHANGED FLAG
70 POKE 6+VARPTR,0:REM CLEAR POSITION IN BUFFER
```

50	FIELD overflow
51	Internal error
52	Bad file number
53	File not found
54	File already open
55	Disk not mounted
56	Disk I/O error
57	File already exists
59	Disk already mounted
61	Input past end
62	Bad file name
63	Direct statement in file
64	Bad allocation table
65	Bad drive number
66	Bad track/sector
67	File write protected
68	Disk offline
69	Deleted record
70	Rename across disks
71	Sequential after PUT
72	Sequential I/O only
73	File not OPEN

H.14 DOUBLE DENSITY, DOUBLE SIDED DISKETTES

For diskettes with 256-byte sectors, DSKI$ and DSKO$ are modified.

The DSKI$ function returns as its value the first 255 bytes of the sector read.

The DSKO$ statement does not use the <string expression> field. The format is:

```
DSKO$ <drive>,<track>,<sector>
```

In order to specify the data to write with DSKO$ and to retrieve all 256 bytes of the data read by DSKI$, the user must FIELD two or more variables (for a total of 256 bytes) to the file#0 buffer. The FIELDed variables will be identical to the data read with DSKI$ and written with DSKO$. For example:

```
FIELD#0,128 AS A$,128 AS B$
```

For double-sided diskettes, the formats of DSKI$ and DSKO$ must also include the surface number:

```
DSKI$(<drive>,<surface>,<track>,<sector>)

DKSO$ <drive>,<surface>,<track>,<sector>
 or
DKSO$ <drive>,<surface>,<track>,<sector>,<string exp>
```

APPENDIX I

Converting Programs to BASIC-80

If you have programs written in a BASIC other than BASIC-80, some minor adjustments may be necessary before running them with BASIC-80. Here are some specific things to look for when converting BASIC programs.

I.1 STRING DIMENSIONS

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the BASIC-80 statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for BASIC-80 string concatenation.

In BASIC-80, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

<u>Other BASIC</u>	<u>BASIC-80</u>
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

<u>Other BASIC</u>	<u>8K BASIC-80</u>
A$(I)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,I+1)
A$(I,J)=X$	A$=LEFT$(A$,I-1);X$;MID$(A$,J+1)
	<u>Ext. and Disk BASIC-80</u>
A$(I)=X$	MID$(A$,1,1)=X$
A$(I,J9=X$	MID$(A$,I,J-I+1)=X$

I.2 MULTIPLE ASSIGNMENTS

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. BASIC-80 would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

I.3 MULTIPLE STATEMENTS

Some BASICs use a backslash (\) to separate multiple statements on a line. With BASIC-80, be sure all statements on a line are separated by a colon (:).

I.4 MAT FUNCTIONS

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

APPENDIX J

Summary of Error Codes and Error Messages

Code	Number	Message
NF	1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
SN	2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).
RG	3	Return without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
OD	4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
FC	5	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: 1. a negative or unreasonably large subscript 2. a negative or zero argument with LOG 3. a negative argument to SQR 4. a negative mantissa with a non-integer exponent

5. a call to a USR function for which the starting address has not yet been given

6. an improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.

OV	6	Overflow The result of a calculation is too large to be represented in BASIC-80's number format. If underflow occurs, the result is zero and execution continues without an error.
OM	7	Out of memory A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.
UL	8	Undefined line A line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE is to a nonexistent line.
BS	9	Subscript out of range An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.
DD	10	Redimensioned array Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.
/0	11	Division by zero A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.
ID	12	Illegal direct A statement that is illegal in direct mode is entered as a direct mode command.
TM	13	Type mismatch A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.

OS 14 Out of string space
String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.

LS 15 String too long
An attempt is made to create a string more than 255 characters long.

ST 16 String formula too complex
A string expression is too long or too complex. The expression should be broken into smaller expressions.

CN 17 Can't continue
An attempt is made to continue a program that:

1. has halted due to an error,

2. has been modified during a break in execution, or

3. does not exist.

UF 18 Undefined user function
A USR function is called before the function definition (DEF statement) is given.

<u>Extended and Disk Versions Only</u>

19 No RESUME
An error trapping routine is entered but contains no RESUME statement.

20 RESUME without error
A RESUME statement is encountered before an error trapping routine is entered.

21 Unprintable error
An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.

22 Missing operand
An expression contains an operator with no operand following it.

23 Line buffer overflow
An attempt is made to input a line that has too many characters.

26 FOR without NEXT
A FOR was encountered without a matching NEXT.

29 WHILE without WEND
A WHILE statement does not have a matching WEND.

30 WEND without WHILE
A WEND was encountered without a matching WHILE.

Disk Errors

50 Field overflow
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

51 Internal error
An internal malfunction has occurred in Disk BASIC-80. Report to Microsoft the conditions under which the message appeared.

52 Bad file number
A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

53 File not found
A LOAD, KILL or OPEN statement references a file that does not exist on the current disk.

54 Bad file mode
An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file or to execute an OPEN with a file mode other than I, O, or R.

55 File already open
A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.

57 Disk I/O error
An I/O error occurred on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.

58 File already exists
The filename specified in a NAME statement is identical to a filename already in use on the disk.

61 Disk full
All disk storage space is in use.

62 Input past end
An INPUT statement is exeucted after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.

63 Bad record number
In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.

64 Bad file name
An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).

66 Direct statement in file
A direct statement is encountered while LOADing an ASCII-format file. The LOAD is terminated.

67 Too many files
An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

APPENDIX K

Mathematical Functions

Derived Functions

Functions that are not intrinsic to BASIC-80 may be calculated as follows.

Function	BASIC-80 Equivalent
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN (X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X-1)) +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X-1)) +(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=EXP(-X)/EXP(X)+EXP(-X))*2+1
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1)
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

[illegible]

APPENDIX M

ASCII Character Codes

ASCII Code	Character	ASCII Code	Character	ASCII Code	Character
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	^
009	HT	052	4	095	<
010	LF	053	5	096	'
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

ASCII codes are in decimal.
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

INDEX